View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by AMS Tesi di Laurea

Alma Mater Studiorum · Università di Bologna

SECONDA FACOLTÀ DI INGEGNERIA

Corso di Laurea Triennale in Ingegneria Elettronica, Informatica e Telecomunicazioni

COORDINATION AS A SERVICE IN JADE

Elaborata nel corso di: Sistemi Distribuiti

Relatore:
Prof. ANDREA OMICINI

Presentata da:
NICOLA DELLAROCCA

Correlatore:
Dott. STEFANO MARIANI

Sessione III
Anno Accademico 2011-2012

Alla mia famiglia

Indice

Elenco delle figure

Elenco dei listati

Introduzione

"*Sono convinto che l'informatica abbia molto in comune con la fisica.*
La differenza, naturalmente, è che mentre in fisica devi capire
come è fatto il mondo, in informatica sei tu a crearlo.
Dentro i confini del computer, sei tu il creatore."
- Linus Torvalds -

Negli ultimi anni ha assunto sempre maggiore importanza il paradigma della programmazione ad agenti, nel quale si concepisce un'applicazione software come un insieme di entità dette, appunto, *agenti* ognuna delle quali ha un proprio obiettivo ed è caratterizzata dal fatto di essere *autonoma*. Un insieme di agenti che sono in grado di interagire fra loro e che vivono in un determinato ambiente è detto *sistema multi-agente* (MAS). Nell'ambito dello sviluppo di questa tipologia di sistemi si colloca *JADE*, un framework sviluppato da Telecom Italia già a partire dal 1998 [16] che può essere utilizzato per la creazione di MAS distribuiti basati su agenti intelligenti e che, anche grazie al fatto di essere *platform-independent*, gode di una notevole diffusione sia in ambito accademico (poiché *open-source*) sia in ambito commerciale (fra le varie aziende aderenti al progetto si può trovare anche Motorola). Dal momento che questi sistemi possono essere composti da grandi quantità di agenti, appare evidente la necessità di possedere meccanismi di coordinazione, i quali sono nati proprio per gestire in maniera corretta le interazioni sociali fra le varie entità che compongono un sistema così come le risorse che esse condividono.

I modelli di coordinazione che, però, sono stati storicamente sviluppati erano concepiti nel contesto di sistemi chiusi, come è il caso di Linda, nato per applicazioni che sfruttano il parallelismo per migliorare le proprie performance [3]. In un tale scenario, dunque, risulta

chiaro come tutte le entità facenti parte del sistema siano note *a priori*, cosa che permette di costruire meccanismi di coordinazione *ad hoc*. Tutto ciò, però, risulta ovviamente inutilizzabile all'interno di sistemi *aperti*, quali possono essere i MAS odierni, per i quali può non essere possible prevedere la quantità e la natura delle entità che saranno presenti ad un determinato momento del run-time. L'obiettivo di questa tesi, quindi, è quello di creare meccanismi di coordinazione basati sul principio di *coordination as a service*, il quale prevede, in contrapposizione alla metodologia vista precedentemente, la creazione di un servizio di coordinazione che possa essere messo a disposizione di un qualunque sistema come se si trattasse di una libreria. Poiché tale servizio non è legato ad un singolo scenario applicativo è necessario permettere la creazione di un'infrastruttura *dinamica*, che possa essere, però, acceduta tramite un insieme di primitive noto *a priori*. A tal proposito si è scelto di utilizzare TuCSoN, un modello di coordinazione tuple-based che nasce con lo scopo di offrire le proprie funzionalità alle entità di un MAS (possibilmente distrubito) attraverso i cosiddetti *centri di tuple*. Questi ultimi, che hanno il ruolo di media di coordinazione e che rappresentano un'evoluzione rispetto agli spazi di tuple presenti in modelli quali Linda, possono essere utilizzati dagli agenti per svolgere tutte le attività di coordinazione di cui necessitano attraverso un insieme limitato e ben definito di primitive.

Sulla base di quanto detto, questo documento prevede, quindi, un primo capitolo di introduzione al concetto di agente e ai sistemi multi-agente, evidenziando come la necessità di meccanismi di coordinazione abbia portato alla creazione di opportuni modelli. Successivamente verranno analizzate le tecnologie che saranno utilizzate nel corso di questa tesi, ovvero JADE e TuCSoN, sottolineando le parti di interesse come, ad esempio, la loro architettura per giungere, infine, all'introduzione del concetto di *coordination as a service*, e alla sua implementazione sulle tecnologie scelte.

Capitolo 1

Coordinazione nei Sistemi Multi-Agente

In questo capitolo si vuole introdurre il lettore ai sistemi multi-agente, in particolare evidenziando la necessità di meccanismi di coordinazione delle entità fondamentali che compongono questa tipologia di sistemi: gli agenti intelligenti. A tal scopo verranno introdotti i modelli di coordinazione a scambio di messaggi e quello tuple-based, ovvero quelli che vengono utilizzati nelle tecnologie che si vedranno in seguito.

1.1 Gli agenti

Con il termine "agente", nonostante non vi sia una definizione accettata universalmente [8], si intende un'entità software che è situata all'interno di un certo *ambiente* e che, per raggiungere i propri obiettivi, è in grado di eseguire delle azioni in maniera autonoma, le quali influiscono sul sopracitato ambiente, modificandolo. In tale "definizione" un ruolo di particolare importanza è rivestito dal termine *autonomo*: con esso si intende la capacità di poter eseguire delle azioni in maniera indipendente. Un agente autonomo, quindi, ha controllo sia sul proprio stato che sul proprio comportamento. In accordo a [5], inoltre, un agente possiede anche le caratteristiche di proattività e socialità, grazie alle quali egli è in grado di iniziare a svolgere le proprie azioni senza la necessità di essere stimolato dall'esterno (ad esempio da parte di esseri umani o di altri sistemi),

e di interagire con altri agenti che coesistono nel medesimo ambiente. Grazie a questi importanti attributi, un agente può essere concepito come un componente software che possiede un determinato comportamento, il quale è strettamente legato agli obiettivi finali dell'agente stesso.

Normalmente all'interno di un sistema si ha una molteplicità di agenti, il cui numero può variare durante l'esecuzione e può non essere noto a priori, cosa che dà vita ad un cosiddetto *sistema multi-agente* (*MAS*). All'interno di un MAS, gli agenti possono svolgere compiti il cui obiettivo può essere comune o contrastante, perciò è possibile che essi interagiscano fra di loro, sia direttamente (attraverso una forma di comunicazione), sia indirettamente (agendo sull'ambiente comune), rispettivamente in maniera costruttiva oppure in competizione.

1.2 Modelli di coordinazione

Le motivazioni che spingono alla creazione di modelli di coordinazione sono da ricercare nella sempre maggiore complessità dei sistemi software, che porta alla proliferazione di entità computazionali indipendenti e, spesso, distribuite. Un modello di coordinazione deve, quindi, fornire una struttura all'interno della quale queste entità, che vengono chiamate "agenti" (con questo termine non si intendono necessariamente gli agenti introdotti nella sezione 1.1), possono esprimere le proprie interazioni, come se il modello stesso fosse una sorta di collante [9, 10]. Il meta-modello proposto da Ciancarini [10] è quello mostrato in Figura 1.1, nella quale si nota un insieme di entità coordinabili (*coordinables*) che possono interagire grazie alla presenza di un *medium di coordinazione*. In accordo a [11], le specifiche di un modello di coordinazione dovrebbero includere:

- Lo spazio di coordinazione, ovvero l'insieme delle entità che possono interagire fra loro. In particolare, lo spazio di coordinazione associato a un modello di coordinazione deve specificare cosa sia un'entità coordinabile ammissibile (ad esempio un processo, un agente, un oggetto), cosa sia un medium di coordinazione (ad esempio un canale di comunicazione, una blackboard, uno spazio di tuple) e cosa sia un evento comunicativo.

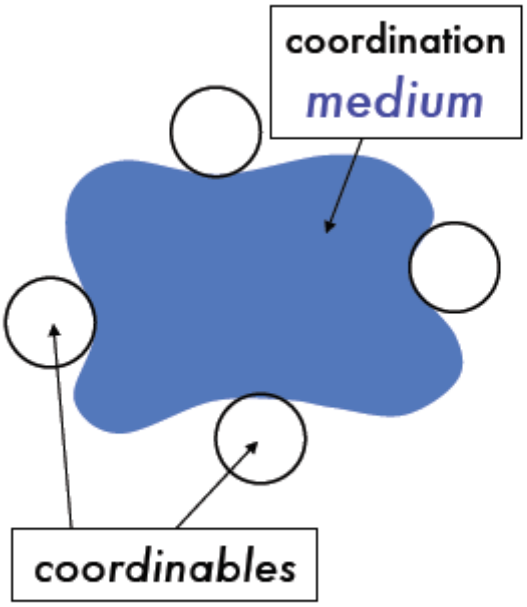


Figura 1.1: Meta-modello di coordinazione

- Il linguaggio di comunicazione, ovvero il linguaggio utilizzato dalle entità per rappresentare un'informazione. Ciò che è importante definire di esso è la sintassi; infatti, poiché si tratta di comunicazione e, quindi, di scambio di informazioni, è necessario che la semantica sia condivisa da tutti i coordinabili, e ciò viene assunto come dato.

- Il linguaggio di coordinazione, ovvero l'insieme di primitive che un coordinabile può invocare; è importante notare che, in questo caso, bisogna specificare anche una semantica per ogni primitiva, definita in termini di "effetto della primitiva sullo spazio di interazione" (ovvero di eventi).

- I media di coordinazione, ovvero quelle entità che hanno il duplice scopo di permettere e favorire la comunicazione fra i coordinabili. Sono, inoltre, caratterizzati dal loro comportamento osservabile, il quale è, a sua volta, regolato dalle leggi di coordinazione.

1.2.1 Coordinazione a scambio di messaggi

Uno dei meccanismi di coordinazione che, storicamente, è stato per primo creato ed utilizzato è quello a *scambio di messaggi* (*message-passing*). Esso rende possibile la comunicazione fra entità di pari livello (peers), ma, non essendo basato su alcun modello di coordinazione, quest'ultima viene raggiunta tramite protocolli che il programmatore è libero di definire. Dal momento che questo "modello" non definisce un preciso linguaggio di comunicazione, è necessario che i peers condividano formato e contenuti dei messaggi, al fine di poter decidere quale comportamento adottare, quindi, ad esempio, se aspettare qualcosa (magari i risultati di una computazione) oppure se fare delle operazioni. Oltre a ciò le entità devono necessariamente essere a conoscenza del *destinatario* di tali messaggi (se necessario, anche più di uno), per cui si ha una forma di accoppiamento fra mittente e ricevente che non è né desiderabile né adatto all'interno di sistemi *aperti*, per i quali non è detto che tutte le entità si conoscano fra loro. L'atto della comunicazione basata su scambio di messaggi è, per di più, transiente, cosa che richiede al destinatario di esistere nel momento in cui egli dovrebbe ricevere il messaggio, altrimenti quest'ultimo potrebbe andare perso nel caso in cui non sia previsto un meccanismo per il suo mantenimento (ad esempio una mail-box).

Un'evoluzione di questo meccanismo di coordinazione consiste nell'utilizzo del cosiddetto *Agent Communication Language* (*ACL*), ovvero un linguaggio di comunicazione standardizzato grazie al quale i contenuti e il formato del messaggio sono definiti in maniera univoca, cosa che promuove lo scambio di informazioni dal momento che tutte le entità in grado di soddisfare tale standard possono comprenderne il significato. Il fulcro degli ACLs è rappresentato dal concetto di *ontologia*, attraverso il quale è possibile specificare il linguaggio ed il vocabolario (insieme alla sua semantica) utilizzati durante l'interazione, in modo da evitare ambiguità e da rendere possibile la comunicazione fra agenti che appartengono a sistemi diversi (purché essi condividano lo stesso ACL).

Se, quindi, da un lato si risolve il problema dell'equivocità dei linguaggi "custom", dall'altro, però, non si introduce nessun modello di coordinazione, per cui rimane il problema visto in precedenza, per la cui risoluzione si rende necessaria l'introduzione di protocolli di comunicazione.

1.2.2 Coordinazione basata su tuple

Per risolvere tali problemi è stato creato il meccanismo di coordinazione basato sulle tuple, che permette di ottenere il grado di disaccoppiamento desiderato. Il linguaggio di comunicazione utilizzato è, come intuibile, quello delle tuple, ovvero collezioni ordinate di elementi informativi non necessariamente omogenei (un esempio di tupla è il seguente `persona('Nicola Dellarocca', 22)`). Oltre a ciò, tale linguaggio definisce anche le cosiddette *anti-tuple o tuple-templates*, grazie alle quali è possibile specificare un insieme di tuple sulla base di un pattern comune, e un meccanismo di *matching* basato su una funzione che permette di ottenere le tuple che corrispondono a un determinato template. Il fulcro di questo modello è, però, rappresentato dal medium di coordinazione, lo *spazio di tuple*, il quale ha il compito di raccogliere tutte le tuple generate dagli agenti, con lo scopo di conservarle e renderle disponibili a chiunque ne abbia necessità. Questo tuple space è accessibile da parte dei coordinabili attraverso il linguaggio di coordinazione che è rappresentato dalle *primitive*, ovvero un insieme di operazioni che permettono di inserire, cercare e consumare delle tuple nel/dal tuple space. Grazie a ciò è possibile permettere l'interazione fra gli agenti senza che essi debbano necessariamente conoscersi: l'unica cosa richiesta è che ognuno di essi possa interagire con il medium di coordinazione tramite le primitive. Inoltre, dal momento che lo spazio di tuple è persistente, si raggiunge anche il disaccoppiamento temporale, cosa che è auspicabile in un sistema distribuito, poiché all'interno di esso non vi è una nozione di "tempo comune" e quindi risulta difficile, se non impossibile, definire *quando* un'azione viene compiuta. Maggiori dettagli su questo modello di coordinazione saranno forniti nel capitolo 3, all'interno del quale verranno anche analizzate alcuni modelli tuple-based.

Capitolo 2

JADE

In questo capitolo si vuole introdurre il lettore a JADE, il middleware per lo sviluppo di agenti e di MAS che verrà utilizzato in questa tesi. Per questo motivo si forniranno alcune nozioni riguardanti gli standard e l'architettura adottati da questa tecnologia.

2.1 JADE come middleware FIPA-compliant

FIPA (Foundation for Intelligent Physical Agents) è un'organizzazione internazionale nata con l'obiettivo di creare un insieme di standards sia per la gestione degli agenti, sia per specificare come gli agenti stessi debbano comunicare ed interagire fra di loro. Dal momento che JADE è FIPA-compliant [5], può essere utile osservare alcuni degli standard definiti, per meglio comprendere l'architettura di JADE stesso.

2.1.1 Gestione degli agenti

Il modello di gestione degli agenti proposto da FIPA [6] è quello mostrato in Figura 2.1. In questo modello si possono notare diverse entità, ognuna delle quali ha un compito ben preciso.

- Agent Platform (AP): si tratta di un contenitore all'interno del quale possono vivere gli agenti.

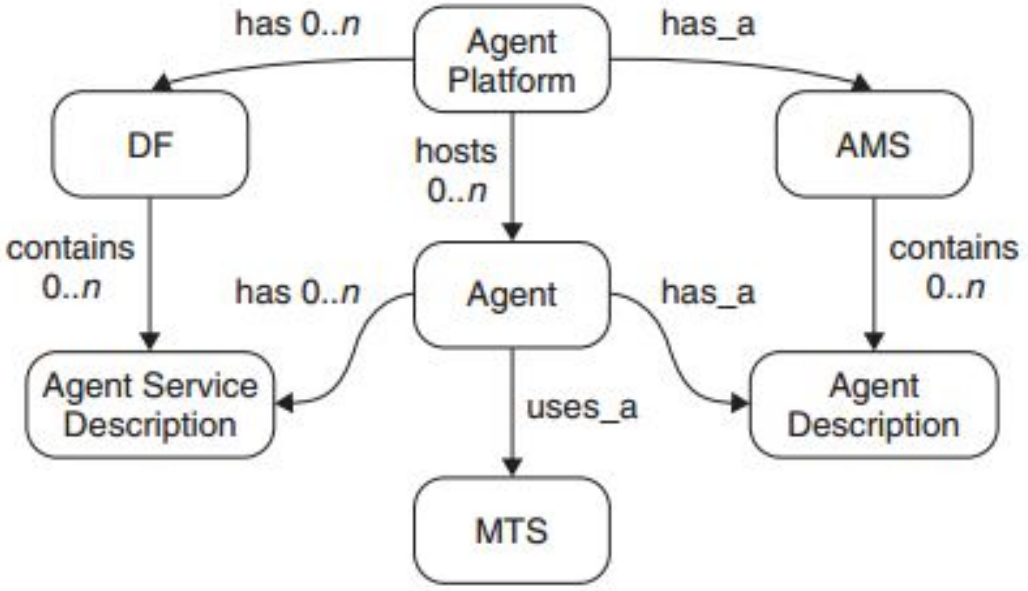


Figura 2.1: Gestione degli agenti

- Agent Management System (AMS): si tratta di un componente fondamentale (come si può notare dall'immagine, ogni AP deve necessariamente avere un AMS). Esso fornisce il servizio di pagine bianche e regola la vita degli agenti all'interno di una piattaforma. Ogni agente, infatti, deve registrarsi presso l'AMS per ottenere un identificatore univoco (AID, Agent ID) e poter essere raggiungibile. Questo identificatore viene poi mantenuto all'interno dell'AMS fino a quando l'agente termina la propria vita (giunto al termine della propria esecuzione, infatti, l'agente si de-registra dall'AMS). Dal momento che l'AMS contiene tutte le descrizioni degli agenti, è possibile interrogarlo per scoprire l'esistenza di un determinato agente, a partire da alcune sue caratteristiche (ad esempio l'AID).

- Directory Facilitator (DF): se l'AMS fornisce il servizio di pagine bianche, il DF fornisce quello di pagine gialle. Presso di esso, infatti, gli agenti possono pubblicare una lista di servizi offerti, lista che può essere interrogata da agenti client.

- Message Transport Service (MTS): fornisce il meccanismo di comunicazione inter-

agente. Si tratta di un servizio fornito dall'AP che supporta il trasporto di messaggi FIPA ACL sia sulla stessa piattaforma, sia sulle altre [7].

2.2 La piattaforma JADE

JADE è una piattaforma open-source che fornisce un middleware per lo sviluppo ed esecuzione di sistemi multi-agente. Essendo totalmente sviluppato in Java, risulta anche platform-independent, e, di conseguenza, adatto allo sviluppo di sistemi distribuiti. Inoltre, data la sua conformità agli standard FIPA, è possibile fare in modo che gli agenti JADE comunichino con qualsiasi altra entità che rispetti tale standard, senza il bisogno dover rispettare altri vincoli particolari. Per quanto riguarda l'architettura di JADE, si può far riferimento alla Figura 2.2. In essa si può notare che ogni agente vive all'interno di un container, il quale è, a sua volta, contenuto in una piattaforma che fornisce alcuni servizi fondamentali come quello di trasporto dei messaggi. Ogni piattaforma è composta da uno o più container, di cui uno è chiamato container principale (main container), ed è quello che contiene, oltre ad eventuali agenti, l'AMS e il DF. Il main container, inoltre, è il primo ad essere avviato all'interno di una piattaforma, ed eventuali altri container devono registrarsi presso di esso. Il fatto che una piattaforma possa contenere più container permette di distribuire sia geograficamente (attivandoli su host diversi) sia concettualmente (attivando più container sulla stessa macchina) il MAS.

2.2.1 Scambio di messaggi in JADE

La comunicazione in JADE è implementata in accordo con lo standard FIPA ACL [4], ed è basata sullo scambio di messaggi asincrono. Ogni agente possiede una casella postale dove vengono recapitati tutti i messaggi destinati a quell'agente. L'unica garanzia che viene data è quella di recapitare il messaggio all'agente corretto (se esiste), mentre è a carico del programmatore decidere se e quando leggere un determinato messaggio. Per facilitare la lettura dei messaggi, inoltre, è possibile ottenere dalla mailbox solamente quelli che soddisfano determinate caratteristiche, specificabili tramite un cosiddetto message-template. Sulla base dello standard FIPA ACL, ogni messaggio contiene diversi campi, alcuni dei quali sono obbligatori:

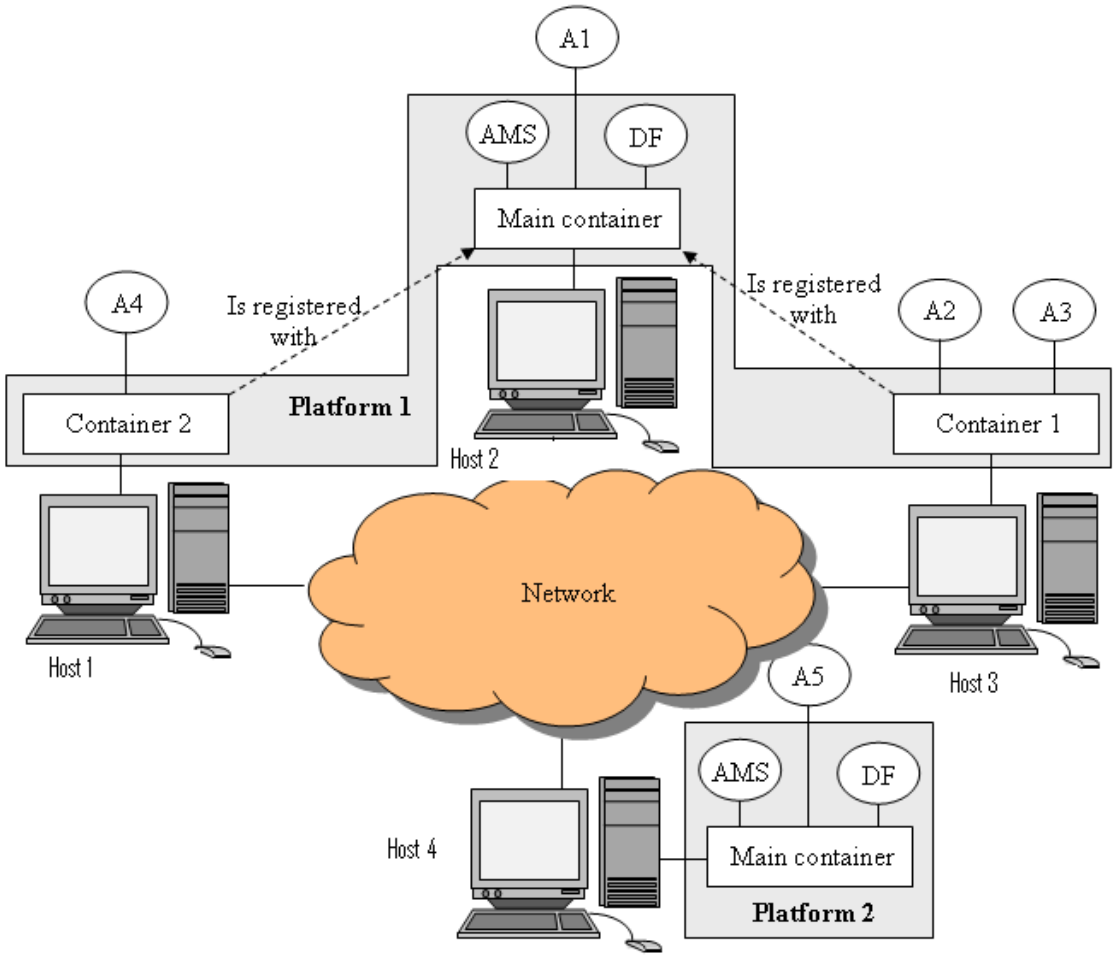


Figura 2.2: Architettura di JADE

- Il *mittente* del messaggio;
- Uno o più *destinatari*;
- Il *performativo* del messaggio; questo concetto deriva dalla teoria degli atti linguistici e indica qual è l'atto che si vuole far eseguire attraverso il contenuto del messaggio.
- Il *contenuto* del messaggio, ovvero l'informazione vera e propria da inviare;

- L'*ontologia* e il *linguaggio*, ovvero l'insieme dei simboli, con i relativi significati, usati all'interno del contenuto. Ovviamente, affinché i vari agenti coinvolti possano comprendere l'informazione trasmessa, è necessario che condividano l'ontologia;

- Altri campi quali *conversation-id*,*reply-to*,ecc... che risultano molto utili in caso di conversazioni multiple e concorrenti.

Per quanto riguarda la comunicazione, la distribuzione è trasparente rispetto al programmatore; infatti, è necessario conoscere solo il mittente del messaggio, senza doversi preoccupare di conoscere la sua posizione, mentre sarà compito del servizio di messaggistica quello di far recapitare il messaggio all'agente desiderato. In questo modo, sempre con riferimento alla Figura 2.2, il programmatore dovrà scrivere esattamente le stesse linee di codice per far comunicare A2 con A3 (agenti nello stesso container), oppure A1 con A2 (agenti sulla stessa piattaforma ma in container diversi) o, infine, per far comunicare A1 con A5 (agenti su piattaforme diverse).

2.2.2 Comportamento di un agente

Come già detto, un agente è un'entità intelligente ed autonoma e, in quanto tale, ha un proprio comportamento, che in JADE è modellato tramite la classe `Behaviour`. In realtà ogni agente può avere anche più comportamenti contemporaneamente, che si susseguono in maniera concorrente. A tal fine ogni agente è dotato di uno scheduler round-robin non-preemptive, sulla base del quale viene scelto quale dei *behaviour*s deve essere messo in esecuzione. Dal momento che lo scheduler è cooperativo, è necessario progettare ogni behaviour in modo da non bloccarne l'esecuzione poiché questo comporterebbe l'impossibilità di passare al behaviour successivo. Per questo motivo, nell'ambito della lettura di messaggi dalla mailbox è stato creato un pattern tale da bloccare l'esecuzione del solo behaviour correntemente in esecuzione fino all'arrivo di un nuovo messaggio, sfruttando il metodo `block()`. Invocando tale metodo, infatti, il comportamento attuale verrà sospeso, e non sarà messo all'interno della coda dello scheduler finché non accadrà un evento tale per cui questo behaviour verrà sbloccato e reinserito nella coda dello scheduler, tipicamente la ricezione di un nuovo messaggio oppure l'invocazione del metodo `restart()` sul behaviour stesso. È importante notare che non è possibile interrompere un

behaviour in un determinato punto e fare in modo che si riprenda l'esecuzione a partire da quel punto in poi, ma ogni volta l'esecuzione ricomincia da capo.

2.3 L'architettura interna di JADE

Le versioni precedenti alla v3.2 prevedevano un kernel JADE monolitico che forniva tutte le funzionalità necessarie per la vita e la comunicazione di un agente JADE. Questo approccio, però, si rivela essere inflessibile poiché richiede modifiche consistenti ogniqualvolta si renda necessario aggiungere nuove funzionalità, con il rischio che tali modifiche vadano a impattare su quelle preesistenti. Dalla versione 3.2 in poi, però, l'architettura di JADE è stata completamente ristrutturata, ottenendone una definita "distributed coordinated filters architecture". A differenza della precedente, essa è stata ideata per favorirne al massimo l'estensibilità e seguendo la strategia "deploy what you need" attraverso la quale vengono messe a disposizione tutte e sole le caratteristiche che saranno effettivamente utilizzate.

In Figura 2.3 è mostrata l'architettura che deriva da questa linea di pensiero. Da essa si può vedere che ogni container risiede al di sopra di un nodo e, così come i container possono ospitare gli agenti, allo stesso modo i nodi possono offrire servizi. È importante evidenziare il fatto che *ogni* container risiede su un nodo diverso, perché questo implica che container diversi, anche se attivi sulla stessa macchina, appartengono a nodi diversi e, quindi, possono offrire servizi diversi. Inoltre, si può notare che vi è un componente chiamato *ServiceManager*, il quale ha il compito di gestire l'attivazione dei servizi all'interno di un nodo e di tener traccia di tutti i servizi che sono attivi su ogni nodo di cui è composta una piattaforma. Ancora una volta il main container riveste un ruolo speciale: se, infatti, all'interno di esso vi è un'istanza del ServiceManager vero e proprio, all'interno di tutti gli altri container della medesima piattaforma vi sono solamente dei *proxy* ad esso.

2.3.1 I servizi kernel

Il componente fondamentale dell'architettura di JADE è, senza dubbio, il *servizio kernel*, poiché tutte le operazioni che un agente può fare sono implementate dal corrispon-

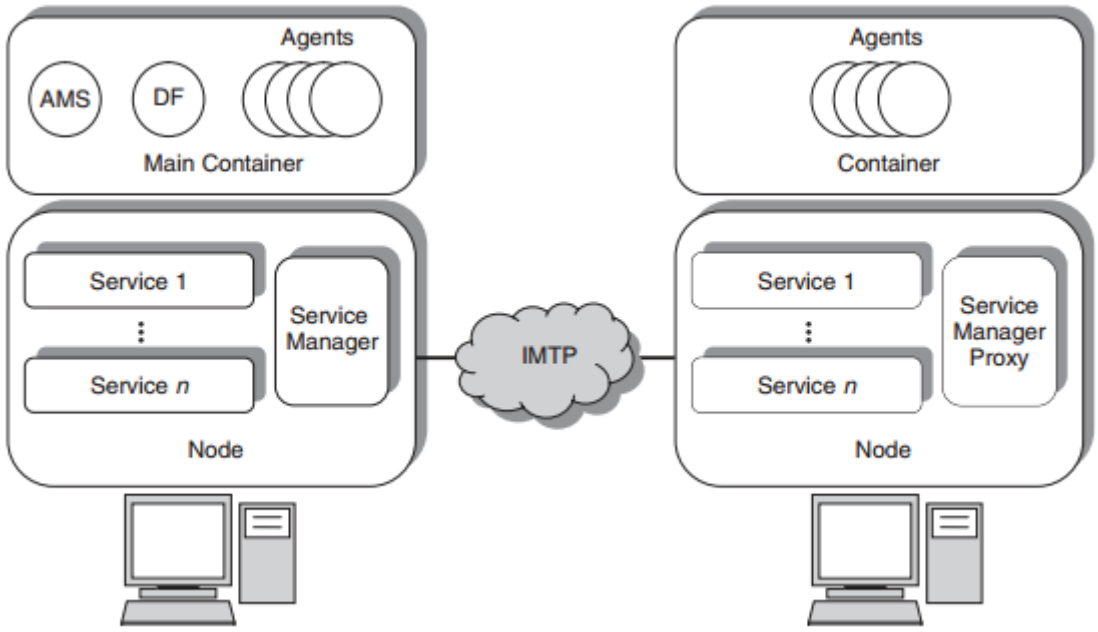


Figura 2.3: Architettura interna di JADE

dente servizio. Ad esempio, l'operazione `send(...)` che consente l'invio di un messaggio ACL da parte di un agente viene eseguita dal servizio di messaggistica, la migrazione verso altri container tramite la `doMove(...)` è a carico dell'AgentMobilityService, e così via.

I comandi verticali

Nello specifico, quando un agente richiede una di queste operazioni, il container inoltra la richiesta al servizio più idoneo per completarla: questo, a sua volta, può implementare direttamente l'operazione oppure creare un cosiddetto *comando verticale*, ovvero un'astrazione per rappresentare l'operazione da svolgere insieme ai parametri necessari per il suo completamento. In quest'ultimo caso il comando verticale attraversa una catena (eventualmente vuota) di "filtri di uscita" (*outgoing filters*) che hanno lo scopo di intercettare il comando ed reagire di conseguenza, eventualmente bloccandolo, per poi arrivare al *source sink*, il componente all'interno del quale si trova l'effettiva implementazione dell'operazione inizialmente richiesta. Per maggiore chiarezza, si consideri la

richiesta da parte di un agente di inviare un messaggio ACL, cosa che genera la seguente catena di eventi (si veda anche la Figura 2.4):

1. L'agente crea il messaggio ACL ed invoca il metodo `send(msg)`.

2. In risposta a ciò, il container sul quale risiede l'agente richiede al MessagingService, se questo è attivo, di inviare il messaggio desiderato.

3. Il MessagingService, a sua volta, non effettua direttamente l'invio ma crea un comando verticale, all'interno del quale inserisce tutti i parametri che sono necessari per la corretta esecuzione dell'operazione (ad esempio il messaggio stesso), dopodiché ne avvia l'elaborazione.

4. Il comando verticale attraversa una catena di filtri di uscita la quale può essere composta, ad esempio, da filtri che permettono di informare componenti quali lo *Sniffer* o l'*Introspector* della richiesta di invio di un messaggio. Eventualmente è possibile creare dei filtri personalizzati che possono anche interrompere l'esecuzione dell'operazione nel caso in cui certi requisiti non siano soddisfatti (ad esempio se l'agente non è autorizzato ad effettuare una certa operazione).

5. Infine, se il comando attraversa con successo l'intera catena, esso viene finalmente eseguito da un componente chiamato *source sink*, che ha il compito di fornire un'implementazione per i comandi verticali che il servizio genera (in questo caso l'implementazione del comando di invio del messaggio consisterà nell'effettiva spedizione di quest'ultimo al destinatario desiderato).

I comandi orizzontali

Un servizio può, inoltre, essere distribuito su più nodi appartenenti alla stessa piattaforma. Riprendendo il servizio di messaggistica, si può notare che esso è, normalmente, installato su tutti i nodi di una piattaforma. Quando il source sink relativo ad un determinato nodo deve processare il comando di invio di un messaggio ad un destinatario che risiede su un container remoto, diventa necessario scoprire su quale degli N possibili nodi esso risieda. Per fare ciò deve, quindi, contattare il servizio di messaggistica

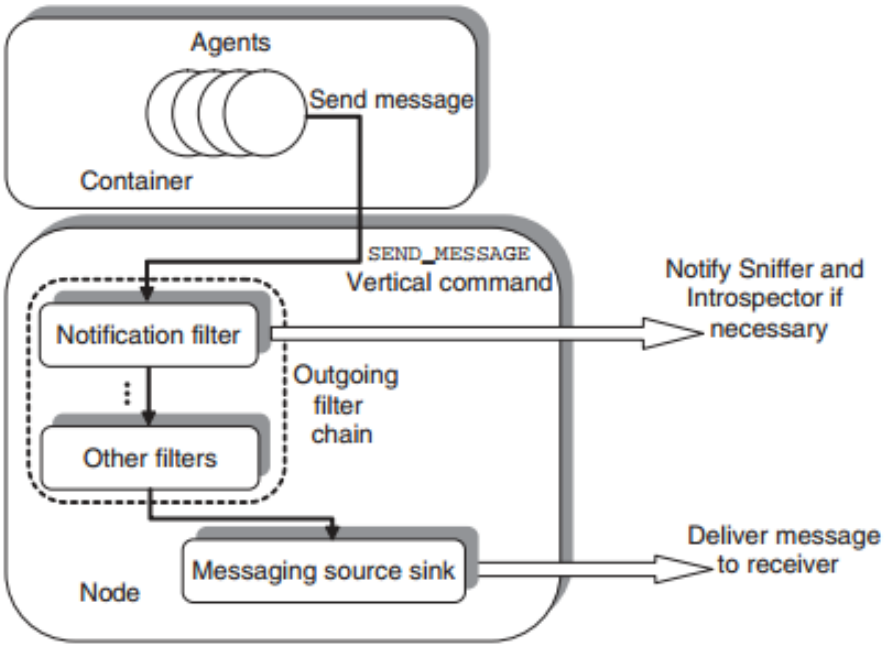


Figura 2.4: L'esecuzione di un comando verticale

che risiede sul main container in modo che quest'ultimo possa interrogare l'AMS, scoprire la posizione di destinazione e restituire il risultato. Inoltre, il nodo di partenza deve contattare il MessagingService sul nodo di destinazione per completare effettivamente il trasferimento, e far recapitare il messaggio all'agente destinatario. Tutte queste operazioni che coinvolgono il medesimo servizio presente su nodi diversi possono essere effettuate grazie ai *comandi orizzontali*, che, in maniera analoga a quelli verticali, incapsulano tutte le informazioni necessare per la loro esecuzione. Queste due tipologie di comandi si completano, dunque, a vincenda, rendendo possibili le interazioni intra-nodo e quelle inter-nodo. Il componente che, all'interno di ogni nodo, è responsabile di inviare i comandi orizzontali è detto *ServiceProxy*, mentre quello che li riceve è detto *slice* (si immagina, infatti, di "affettare" il servizio, distribuendo ogni fetta su un nodo diverso). In maniera analoga a quanto descritto precedentemente, uno slice può implementare direttamente l'operazione oppure può affidarla ad un sink, con la relativa catena di filtri. In questo caso la loro nomenclatura è, rispettivamente, *target sink* e *incoming filters*. L'esecuzione di un comando orizzontale è schematizzata in Figura 2.5.

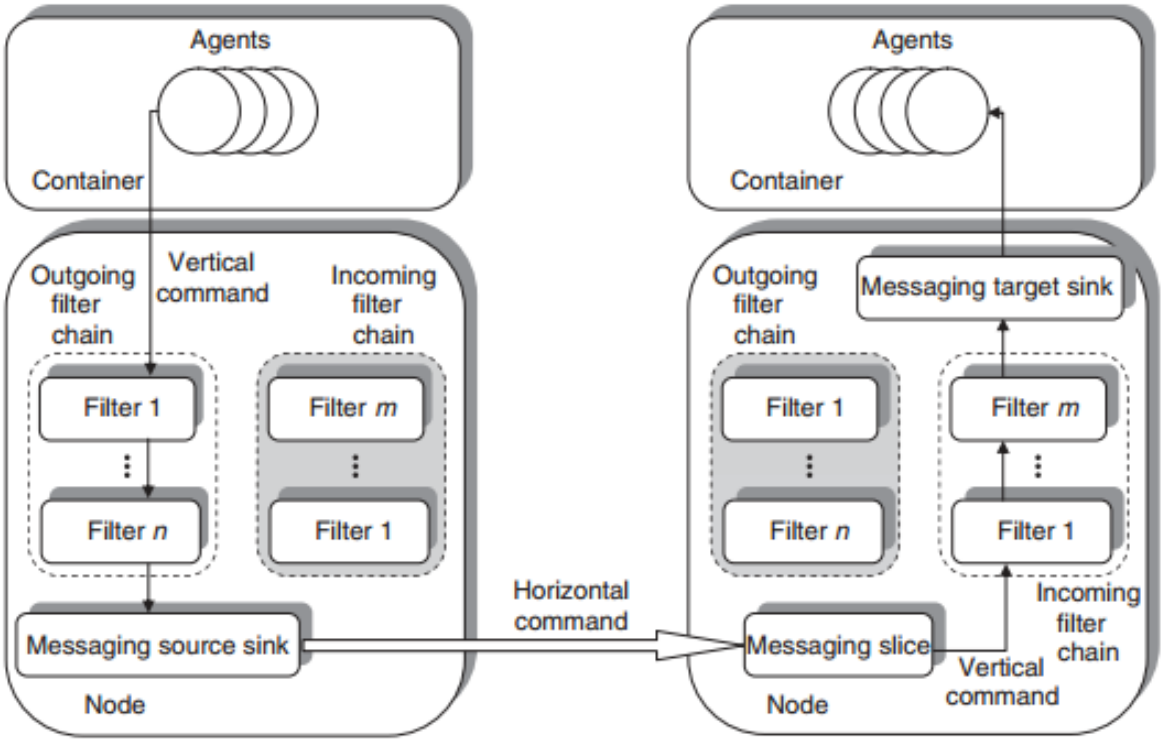


Figura 2.5: L'esecuzione di un comando orizzontale

I service helpers

Mentre per i servizi nativamente presenti in JADE le operazioni sono generalmente gestite dal container, per i servizi custom ciò non avviene. Per permettere, quindi, ad un agente di utilizzare le operazioni che un servizio mette a disposizione sarebbe, dunque, necessario modificare i sorgenti del container. Dal momento che questo era uno dei problemi che hanno spinto a modificare l'architettura di JADE, si è introdotto il concetto di *service helper*, considerato come entità il cui compito è quello di fare da tramite fra l'agente ed il servizio stesso. Attraverso di esso, inoltre, è possibile modificare in maniera significativa il kernel pur non mettendo mano ai sorgenti: un agente può, infatti, ottenere l'helper relativo ad un servizio tramite il metodo `getHelper(...)` e, successivamente, utilizzarlo per sfruttare le operazioni messe a disposizione dal relativo servizio.

Capitolo 3

TuCSoN

All'interno di questo capitolo si vogliono fornire alcune informazioni utili per la comprensione della tecnologia TuCSoN, che verrà utilizzata successivamente. A tal fine, viene introdotto Linda come esempio di modello di coordinazione tuple-based, poiché è da una sua evoluzione che nasce il modello TuCSoN.

3.1 Linda: un modello di coordinazione Tuple-based

Si analizza ora un modello tuple-based concreto, nello specifico il modello Linda [13], poiché è da esso che deriva TuCSoN. Per ciò che concerne il linguaggio di comunicazione è stato scelto dagli inventori quello delle *tuple* per lo scambio di informazioni, quello delle *anti-tuple* o *tuple-templates* per la loro ricerca sulla base di un pattern e, infine, un meccanismo di *matching* che permette di stabilire se una determinata tupla appartiene o meno ad un determinato template. Tutte le tuple vengono "raccolte" all'interno di un cosiddetto *spazio di tuple*, che riveste il ruolo di medium di coordinazione. Le primitive per interagire con tale entità sono diverse e possono riferirsi alle tuple (T) o ai tuple-templates (TT). Le primitive disponibili nel modello originale di Linda sono le seguenti:

- `out(T)`, che inserisce la tupla `T` nello spazio di tuple.
- `in(TT)`, che restituisce al chiamante una qualsiasi tupla `T` che faccia match con il template `TT`, rimuovendola anche dal tuple-space. Ha una semantica sospensiva,

per cui se lo spazio di tuple non contiene nessuna tupla `T*` che sia compatibile con `TT` allora l'operazione viene sospesa per poi essere completata quando `T*` sarà disponibile. Inoltre, nel caso vi siano più tuple in accordo con il template, al chiamante ne verrà restituita una sola, scelta in modo non deterministico.

- `rd(TT)`, che è identica alla precedente ad eccezione del fatto che la lettura non è distruttiva, e quindi la tupla `T*` non viene cancellata dallo spazio di tuple.

Dal momento che tali primitive possono essere utili per la risoluzione di un sottoinsieme limitato di problemi, in breve tempo si è avuta la necessità di introdurne di nuove per aggiungere nuove funzionalità, modificando le interazioni possibili fra i coordinabili e lo spazio di tuple. È questo il caso delle primitive *predicative* (`inp`, `rdp`) e delle *Bulk* (`in_all`, `rd_all`), in cui le prime introducono il concetto di successo o fallimento di una primitiva eliminando allo stesso tempo la semantica sospensiva, mentre le seconde introducono la possibilità di effettuare operazioni su gruppi di tuple che rispettano uno stesso pattern. L'utilizzo di un solo tuple space per la gestione di tutte le tuple, tuttavia, rappresenta una centralizzazione, con tutti i possibili problemi di performance e fault-tolerance che ciò comporta. Per questo motivo, un'ulteriore evoluzione di Linda prevede l'utilizzo di *molteplici* spazi di tuple, eventualmente distribuiti, che sono resi accessibili ai coordinabili grazie alla sintassi `ts@node ? operation`, con la quale si richiede l'esecuzione della primitiva `operation` sul tuple space `ts` situato sul nodo `node`.

Sulla base di quanto visto finora, possiamo dire che il modello Linda gode delle seguenti proprietà:

- Comunicazione generativa: le tuple non sono legate alle entità che le hanno create e sono ugualmente accessibili da tutti i coordinabili. Il loro mantenimento all'interno degli spazi di tuple permette di avere disaccoppiamento spazio-temporale fra gli agenti.

- Accesso associativo: le tuple vengono accedute all'interno di un tuple space sulla base del loro contenuto e struttura, piuttosto che dal loro nome o posizione.

- Semantica sospensiva: la coordinazione è resa possibile grazie alla semantica sospensiva di alcune primitive, per cui è, di fatto, strettamente collegata alla presenza o meno di determinate informazioni all'interno di un tuple space.

3.2 Verso i centri di tuple

Il *modus operandi* appena visto che ha portato all'introduzione di nuove primitive per ogni problema altrimenti irrealizzabile, si è, però, dimostrato poco valido, in quanto non porta alla realizzazione di soluzioni *general-purpose* e rende sempre più difficile garantire quella caratteristica di "apertura" che, invece, è richiesta nell'ambito di sistemi aperti. Questo, infatti, può portare all'impossibilità da parte di un'entità di entrare a far parte di un sistema in quanto esso utilizza delle primitive definite *ad hoc* che l'entità può non conoscere. La causa principale di questi problemi è da attribuire alla staticità che caratterizza il comportamento dello spazio di tuple, i quali hanno un comportamento fissato e immutabile che può essere adatto per la risoluzione di alcune tipologie di problemi ma non per altre. Una possibile soluzione, alternativa a quella descritta precedentemente che consiste nell'introduzione di nuove primitive, potrebbe essere quella di implementare le politiche di coordinazione non a livello di tuple space, ma direttamente all'interno dei coordinabili. Tale scelta, però, si rivela essere decisamente scorretta, poiché non fa altro che "spostare" la logica di coordinazione dai tuple spaces ai coordinabili, in opposizione a quella che è la vera natura dei coordination media.

Ciò che si vuole ottenere, invece, è un insieme di media di coordinazione il cui comportamento possa essere modificato dinamicamente, in modo che essi possano essere adattati agli scenari più disparati. Si tratta, quindi, di creare un modello ibrido, che contenga in sé sia un livello *information-driven* come in Linda (la coordinazione avviene sulla base della presenza/assenza di determinate informazioni) sia uno *action-driven* in modo tale da rendere possibile l'osservabilità degli eventi e quindi anche la possibilità di reagire ad essi.

3.2.1 I centri di tuple

Per questo motivo si introduce l'astrazione di *centro di tuple*, ossia uno spazio di tuple arricchito della possibilità di definire dinamicamente il proprio comportamento in risposta a determinati eventi. Per definire tale comportamento è necessario utilizzare un opportuno *reaction specification language*, il quale permette di definire un insieme di attività computazionali (dette *reazioni*) che devono essere svolte in risposta ad un deter-

minato evento che avviene sul tuple centre. Nello specifico, ogni reazione può effettuare una qualsiasi serie di computazioni che comprenda una o più delle seguenti operazioni: modifica dello stato del tuple centre (aggiungendo/rimuovendo delle tuple), accesso alle informazioni relative all'evento che scatena l'azione (l'evento viene reificato, quindi è reso completamente osservabile) oppure invocazione di primitive su altri tuple centre. Come conseguenza, la semantica delle primitive di coordinazione non è più vincolata ad essere come in Linda, ma può raggiungere un grado di complessità tale da soddisfare i requisiti che alcuni sistemi possono richiedere. È importante notare che, se l'insieme di reazioni è vuoto, il centro di tuple è, a tutti gli effetti, equivalente ad uno spazio di tuple. Dal punto di vista del coordinabile, il tuple centre è sempre percepito come se fosse un tuple space, con la differenza che in quest'ultimo le transizioni di stato sono fissate uguali per tutti gli scenari applicativi (una out porterà sempre all'immissione di una tupla), mentre nel primo le transizioni possono essere "personalizzate" per renderle application-specific. Rimane, comunque, inalterata la percezione di atomicità che caratterizza le primitive (un coordinabile non può rendersi conto di eventuali computazioni intermedie in atto da parte del tuple centre) e questo è dovuto anche alla semantica *transazionale* delle reazioni, cosa che porta al *rollback* dello stato del tuple centre in caso di fallimento di una delle operazioni previste come reazione ad un evento.

3.2.2 ReSpecT

ReSpecT rappresenta un'implementazione del reaction specification language appena introdotto. Un centro di tuple ReSpecT prevede, in realtà, due spazi ben separati: lo spazio delle tuple *ordinarie*, atto a contenere le normali tuple di informazione tramite le quali i coordinabili possono interagire e che equivale al tuple space di Linda, e lo spazio delle tuple di *specifica*, ovvero quelle tuple di reazione in grado di modificare il comportamento del tuple centre e, di conseguenza, le operazioni di coordinazione. I centri di tuple ReSpecT adottano per entrambi gli spazi appena descritti le tuple logiche, che rappresentano un'implementazione per il linguaggio di comunicazione. In particolare, per le tuple ordinarie vengono utilizzate i predicati appartenenti alla logica del primo ordine, mentre per le tuple di specifica vengono utilizzate particolari tuple logiche espresse nella forma `reaction (ET, G, R)`. Con tale tupla si intende dire che al verificarsi di un evento

E compatibile con il template ET, se la guardia G (che è opzionale) è vera allora viene scatenata la reazione R, altrimenti se la guardia è falsa non viene eseguita alcuna reazione.

I centri di tuple ReSpecT godono di alcune proprietà fonamentali:

- Ispezionabilità: entrambi gli spazi di cui sono composti i centri di tuple risultano ispezionabili da parte degli agenti tramite le operazioni di rd e no per lo spazio ordinario, e tramite le rd_s e no_s per lo spazio di specifica. Per tale compito, invece, gli amministratori possono sfruttare strumenti come l'Inspector.

- Malleabilità: il comportamento del tuple centre è definito sulla base delle tuple presenti nello spazio di specifica, per cui può essere adattato modificando tale spazio attraverso le operazioni di in_s e rd_s.

- Linkabilità (linkability): è possibile creare tuple di specifica che permettano di creare reazioni in grado di eseguire primitive di coordinazione su un qualsiasi centro di tuple presente sulla rete.

3.3 Il modello di coordinazione TuCSoN

TuCSoN[1] è un modello di coordinazione tuple-based da cui deriva la tecnologia TuCSoN che sarà utilizzata in questa tesi e che può essere utilizzata per la coordinazione di agenti intelligenti distribuiti e, possibilmente, mobili. Le entità principali che caratterizzano questo modello sono le seguenti:

- Agenti TuCSoN: si tratta dei coordinabili. Dato che possono essere mobili, è possibile che migrino da un dispositivo ad un altro.

- Centri di tuple ReSpecT: assumono il ruolo di coordination media. Sono vincolati ad esistere sullo stesso dispositivo, per cui la loro mobilità è strettamente legata alla mobilità del dispositivo (ad esempio uno smartphone Android).

- I nodi TuCSoN: sono l'astrazione topologica di base ed ospitano i centri di tuple.

[1] **Tu**ple **C**entres **S**pread **o**ver the **N**etwork. http://alice.unibo.it/xwiki/bin/view/TuCSoN/.

Dal momento che in un sistema distribuito possono esistere molteplici istanze per ognuna delle entità appena descritte, è necessario utilizzare un sistema di naming. Per quanto riguarda i nodi, l'identificatore è `netId:port`, dove `netId` può essere un indirizzo IP oppure un nome DNS, mentre `port` è il numero di porta. Per gli agenti, invece, viene utilizzato come nome `agentName` un termine Prolog ground (senza variabili) seguito da un UUID[2] (Universally Unique ID) il quale viene assegnato automaticamente dal middleware all'agente una volta che quest'ultimo inizia ad usarlo. Infine, i centri di tuple sono identificati dalla struttura vista precedentemente (`tc@node`), dove `tc` è il nome del tuple centre (anch'esso un termine ground) mentre `node` rappresenta l'identificatore del nodo appena descritto. La sintassi completa è, quindi, la seguente: `ts@netId:port`. Il motivo per cui è necessario specificare l'identificatore del nodo è evidente: diversi nodi TuCSoN possono ospitare centri di tuple il cui nome `tc` può essere lo stesso.

Dal momento che non si pongono particolari limiti sul nome `tc`, in linea puramente teorica un nodo TuCSoN potrebbe ospitare infiniti centri di tuple. Dato che ciò è evidentemente irrealizzabile, in TuCSoN si distingue fra tuple centres *potenziali* e *attuali*. Con i primi si intendono tutti i tuple centres ammissibili, mentre con i secondi quelli effettivamente utilizzati. Inizialmente, all'avvio del nodo, tutti i centri di tuple rientrano nella prima categoria; alla prima richiesta di operazioni su un centro che non è attuale vengono allocate tutte le risorse necessarie per il suo mantenimento, rendendolo attuale.

3.3.1 Primitive di coordinazione

Dal momento che TuCSoN utilizza i centri di tuple ReSpecT come media di coordinazione, il linguaggio di comunicazione utilizzato è quello, precedentemente introdotto, delle tuple logiche. Riguardo al linguaggio di coordinazione, invece, TuCSoN possiede tutte le primitive già viste in Linda, mantenendone la semantica. In aggiunta ad esse vengono introdotte le operazioni `no` e `nop` per verificare l'assenza di determinate tuple, `set` e `get` per impostare e ottenere l'intero spazio di tuple ordinarie. Per ognuna di queste operazioni sono rese disponibili anche le relative controparti per lo spazio di specifica. Per una lista più dettagliata si veda l'Appendice A.

[2] `http://docs.oracle.com/javase/7/docs/api/java/util/UUID.html`.

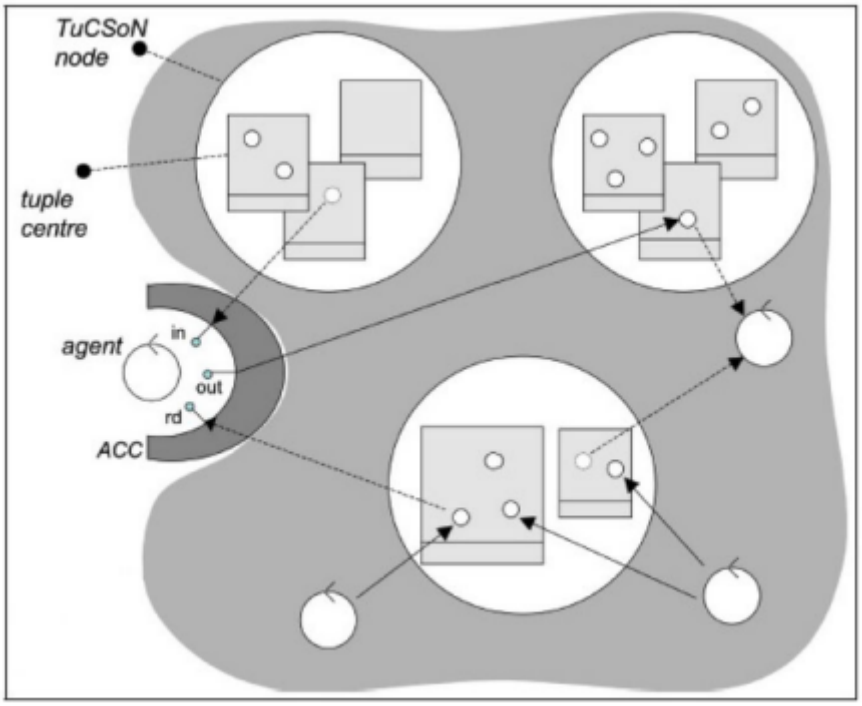


Figura 3.1: L'ACC come interfaccia

3.3.2 L'Agent Coordination Context

L'Agent Coordination Context (ACC) può essere concepito come una sorta di interfaccia che l'infrastruttura fornisce ad un agente per permettergli e allo stesso tempo vincolarlo ad effettuare tutte e sole le interazioni che l'ACC stesso permette. È opportuno sottolineare che il termine *interfaccia* può essere fuorviante, poiché in ambito object-oriented con esso si intende un contratto che un'oggetto dovrà poi rispettare, fornendo una visione dall'esterno dell'oggetto stesso. In questo ambito, invece, l'ACC è da intendere come un'interfaccia nel senso che attraverso di esso un agente può interfacciarsi con il mondo TuCSoN. Tramite esso si può decidere a quale livello un agente possa avere percezione dello stato dei centri di tuple, così come il suo grado di modifica di tale stato. Non solo: il sistema stesso non percepisce direttamente l'agente, ma solamente il suo ACC, che, in questo senso, può essere considerato un tramite (Figura 3.1).

Il motivo per cui viene introdotto un ACC è quello di fornire una visione personalizzata del sistema all'utilizzatore, limitando al tempo stesso le interazioni che l'utente può

effettuare con il sistema. Il meccanismo di rilascio dell'ACC dovrebbe avvenire a seguito di una negoziazione fra agente e sistema, in seguito alla quale sarà definito l'insieme di operazioni che il primo sarà in grado di effettuare sul secondo. Per questo motivo, quindi, un ACC può essere un'astrazione di *organizzazione* (nel suo significato di "ente") adatta per modellare RBAC in TuCSoN [14]. Sulla base di ciò, l'ACC si mostra essere un'entità all'interno della quale è possibile incapsulare politiche di sicurezza basate sul riconoscimento dell'appartenenza di un agente ad un determinato ruolo. Per fare un esempio, si può immaginare l'esistenza di un ruolo di *lettore*, al quale viene associata la possibilità di effettuare determinate operazioni (ad esempio di lettura dello stato del sistema) ma non altre (ad esempio di modifica dello stato del sistema), che magari possono essere associate ad altri ruoli (ad esempio *scrittore*); in questo scenario, un agente può negoziare con il sistema il rilascio di un ACC per il ruolo di lettore, cosa che porterebbe l'agente a "vederlo" come se esso fosse un'entità immutabile (dal momento che non è possibile modificarne lo stato).

3.3.3 L'ACC in TuCSoN

Nel modello TuCSoN è implementato il concetto di ACC. Un agente non può, infatti, interagire *direttamente* con i tuple centres, ma può farlo solamente tramite il proprio ACC, il quale, facendo da interfaccia, fornisce un elenco di tutte le operazioni che l'agente può invocare. In realtà nella versione attuale di TuCSoN non è previsto un meccanismo di negoziazione dell'ACC in quanto il sistema rilascia sempre un un `EnhancedACC`, ovvero l'ACC che mette a disposizione tutte le possibili primitive di coordinazione. Per questo motivo è il programmatore che può decidere a design-time e in maniera statica di limitare i privilegi dell'agente, decidendo quale "ruolo" egli assumerà.

Capitolo 4

Coordination as a Service

Le due tecnologie precedentemente introdotte, JADE e TuCSoN, sono attualmente scorrelate e indipendenti: questo significa che un MAS sviluppato in JADE non può sfruttare i servizi di coordinazione messi a disposizione da TuCSoN, se non attraverso l'uso delle API che quest'ultimo offre. Lo scopo di questa tesi è, quindi, quello di fornire agli agenti JADE un meccanismo tale per cui l'interazione con TuCSoN possa avvenire in maniera più semplice e ad un livello di astrazione maggiore. Tale meccanismo deriva dal concetto di "Coordination as a Service" (CaaS).

4.1 Il concetto di CaaS

A tal scopo è utile la nozione di "Coordination as a Service", in contrasto con quella di "Coordination as a Language" (CaaL) [3]. Quest'ultima nasce insieme a modelli di coordinazione come Linda, che erano concepiti per essere utilizzati all'interno di sistemi chiusi, in cui tutte le entità erano conosciute già a *design-time*, cosa che permette di creare dei media di coordinazione molto performanti per il singolo scenario applicativo ma non adatti ad un caso generale rendendoli, di fatto, parte dell'applicazione stessa. Il punto focale di questo concetto è rappresentato dalle primitive, mentre lo spazio di tuple riveste un ruolo marginale, avendo la sola funzione di "raccoglitore" di tuple. Un programmatore per un sistema di questo tipo deve, quindi, creare delle operazioni che abbiano la stessa semantica delle primitive, eventualmente modificandola per adattarla

alla situazione di interesse, e un'infrastruttura statica composta da tutti e i soli tuple spaces di cui necessita. L'evoluzione dell'informatica ha, però, portato alla nascita di sistemi con complessità sempre maggiore e con la necessità di *apertura*, cosa che rende l'approccio precedentemente descritto inadatto ad essere utilizzato per questa nuova tipologia di sistemi. Per questo motivo è nato il concetto di "Coordination as a Service", con il quale si pone l'attenzione sul medium di coordinazione, che, a differenza della CaaL, non è più statico ma può essere modificato aggiungendo o rimuovendone degli elementi. L'idea che sta alla base di esso è quella di fornire a qualsiasi sistema ne abbia bisogno, dei meccanismi che sono stati creati appositamente per la coordinazione e che hanno il notevole vantaggio di essere general-purpose e, quindi, adatti a qualsiasi scenario applicativo. Nello specifico, l'obiettivo di questa tesi è quello di creare un vero e proprio servizio di coordinazione, in modo da rendere disponibile a chiunque decidesse di utilizzarlo un meccanismo diverso da quello a scambio di messaggi che è nativamente presente all'interno di questa piattaforma.

Per meglio chiarire il concetto, si può fare riferimento alla sua analogia con il servizio di trasporto pubblico, nel quale i servizi offerti sono noti a priori e ben definiti (ad esempio la linea numero 1 percorre un determinato itinerario, effettuando alcune fermate che sono note già prima di salire sul mezzo stesso); inoltre, tali servizi sono messi a disposizione a prescindere dall'effettiva richiesta che ci sarà (ad esempio un autobus viaggia anche se non trasporta nessun passeggero, perché qualcuno potrebbe salire in futuro).

4.2 Scelte architetturali

Per realizzare tutto ciò si può sfruttare l'architettura di JADE che rende esplicito il concetto di "servizio" come modalità per aggiungere funzionalità a run-time sfruttando i cosiddetti kernel services. Grazie ad essi un agente JADE sarà in grado di utilizzare TuCSoN come se quest'ultimo fosse stato implementato direttamente all'interno della piattaforma JADE.

Poiché il servizio TuCSoN da sviluppare sarà l'unica interfaccia che un agente JADE avrà con il mondo TuCSoN, è opportuno valutare con attenzione quali dovranno essere le funzionalità che tale servizio dovrà offrire. Analizzando la tecnologia TuCSoN emerge

che l'entità fondamentale per l'utilizzatore è l'ACC (Agent Coordination Context). È solo tramite esso, infatti, che è possibile eseguire le operazioni primitive, per cui risulta evidente che il servizio TuCSoN dovrà, in qualche maniera, permettere agli agenti di ottenere e rilasciare un ACC, in modo da rendere possibile l'interazione con i centri di tuple. In una prima analisi, di fronte a questo vincolo si hanno due alternative: la prima consiste nell'affidare l'ACC direttamente nelle mani dell'agente che lo richiede, in modo che quest'ultimo possa poi eseguire le operazioni direttamente su tale oggetto, mentre la seconda consiste nel gestire l'ACC in modo trasparente rispetto all'agente, rendendo il servizio una sorta di proxy di TuCSoN. Approfondendo lo studio di queste due possibili scelte, però, si nota come esse siano, a loro modo, "estreme": nella prima, infatti, il servizio TuCSoN rivestirebbe un ruolo marginale, avendo il solo compito di permettere all'utente di ottenere e rilasciare il coordination context, delegando all'agente il compito di interagire con esso, peraltro allo stesso livello di astrazione che si avrebbe utilizzando direttamente le API di TuCSoN. Con la seconda scelta, invece, si avrebbe la situazione opposta, per cui l'agente interagirebbe esclusivamente con il servizio per effettuare ogni operazione, con lo svantaggio di centralizzare tutte le responsabilità nel servizio, cosa che potrebbe causare un "collo di bottiglia". Di conseguenza, noti i vantaggi e gli svantaggi di entrambe le scelte, si è optato per una soluzione "ibrida", che possa permettere di gestire l'ACC in maniera trasparente per l'agente pur mantenendo il servizio snello e reattivo. Per raggiungere tale obiettivo, si è deciso di creare un servizio minimale, che esponga un'interfaccia limitata in grado di offrire poche operazioni, quali:

- Autenticazione (o deautenticazione): questa operazione permette all'agente che la richiede di effettuare l'autenticazione (o deautenticazione). In realtà la versione attuale di TuCSoN[1] non prevede alcun meccanismo di autenticazione né di sicurezza, per cui questa operazione è, di fatto, triviale e porta sempre all'acquisizione (o rilascio) di un *EnhancedACC*, ovvero l'ACC più vasto possibile. Attraverso essa, però, si introduce la possibilità di affrontare la questione della sicurezza direttamente a livello di agente piuttosto che di middleware TuCSoN.

- Lancio (o spegnimento) di un nodo TuCSoN: questa operazione permette di avviare

[1] TuCSoN-1.10.3.0206.

(o fermare) un nodo TuCSoN sulla macchina locale. Tale nodo sarà a tutti gli effetti un nodo TuCSoN, e potrà essere utilizzato da chiunque (agenti JADE, agenti TuCSoN, ...).

- Ottenimento di un TucsonOperationHandler: questa operazione è la più importante fra quelle elencate, in quanto permette di ottenere un oggetto tramite il quale sarà possibile invocare le primitive su un qualunque tuple centre (sia esso locale o remoto) senza dover chiamare in causa il TuCSoNService.

4.2.1 Il TuCSoNHelper

Poiché, come visto precedentemente, per rendere accessibile un servizio agli agenti bisogna creare il relativo helper, dal momento che si crea un TuCSoNService si rende automaticamente necessario creare anche il TuCSoNHelper. Poiché è esclusivamente tramite di esso che gli agenti possono utilizzare il servizio, è necessario che l'interfaccia di questo helper metta a disposizione tutte quelle operazioni che erano state precedentemente indicate come "fondamentali".

4.2.2 Il TucsonOperationHandler

Il `TucsonOperationHandler` nasce come risposta all'esigenza di trasparenza per quanto riguarda l'ACC. Infatti, ciò che è importante dal punto di vista di un agente JADE è il fatto di eseguire una primitiva su un determinato centro di tuple, senza doversi occupare di *come* dover eseguire tale operazione. Analizzando, dunque, la sintassi richiesta da TuCSoN per l'esecuzione di una primitiva, è possibile isolarne le parti fondamentali. Una generica richiesta ad un tuple centre sarà rappresentata dalla stringa `tupleCentreName@NODE_IP:NODE_PORT_NO ? primitive(tuple)`, in cui è possibile distinguere 3 termini principali: un identificatore univoco per il tuple centre target (composto dal nome del tuple centre e dal suo indirizzo IP completo di numero di porta), il tipo di primitiva e la tupla oggetto della primitiva. L'intento del TucsonOperationHandler è, quindi, proprio quello di permettere all'utilizzatore di interagire con il tuple centre conoscendo solamente le informazioni indispensabili, e gestendo in maniera trasparente tutti

quegli aspetti che non sono fondamentali per l'agente ma sono richiesti dalla tecnologia, come ad esempio l'ACC.

Per ottenere il disaccoppiamento desiderato si è, dunque, spostata l'attenzione dall'ACC al tipo di primitiva, creando l'entità `TucsonAction`, avente lo scopo di rappresentare una generica primitiva TuCSoN. Analizzando le numerose primitive che TuCSoN mette a disposizione, si possono notare due classi ben distinte di operazioni: quelle ordinarie e quelle di specifica, in cui le prime vanno a definire il *contenuto* del tuple centre, mentre le seconde, molto più potenti, possono modificarne il `comportamento`. Risulta evidente come queste ultime debbano essere gestite con maggiore attenzione, per evitare che agenti malevoli possano mettere in pericolo il normale funzionamento del tuple centre (ad esempio facendo in modo che tutte le primitive non abbiano effetto sul tuple centre, cosa che renderebbe quest'ultimo inutilizzabile). In Figura 4.1 è mostrata la gerarchia di operazioni appena descritta.

Sulla base di ciò è possibile costruire il `TucsonOperationHandler` come un oggetto in grado di eseguire ognuna di queste operazioni sia in maniera sincrona che asincrona.

Il rapporto che esiste fra questo oggetto ed il TuCSoNService è descritto in Figura 4.2.

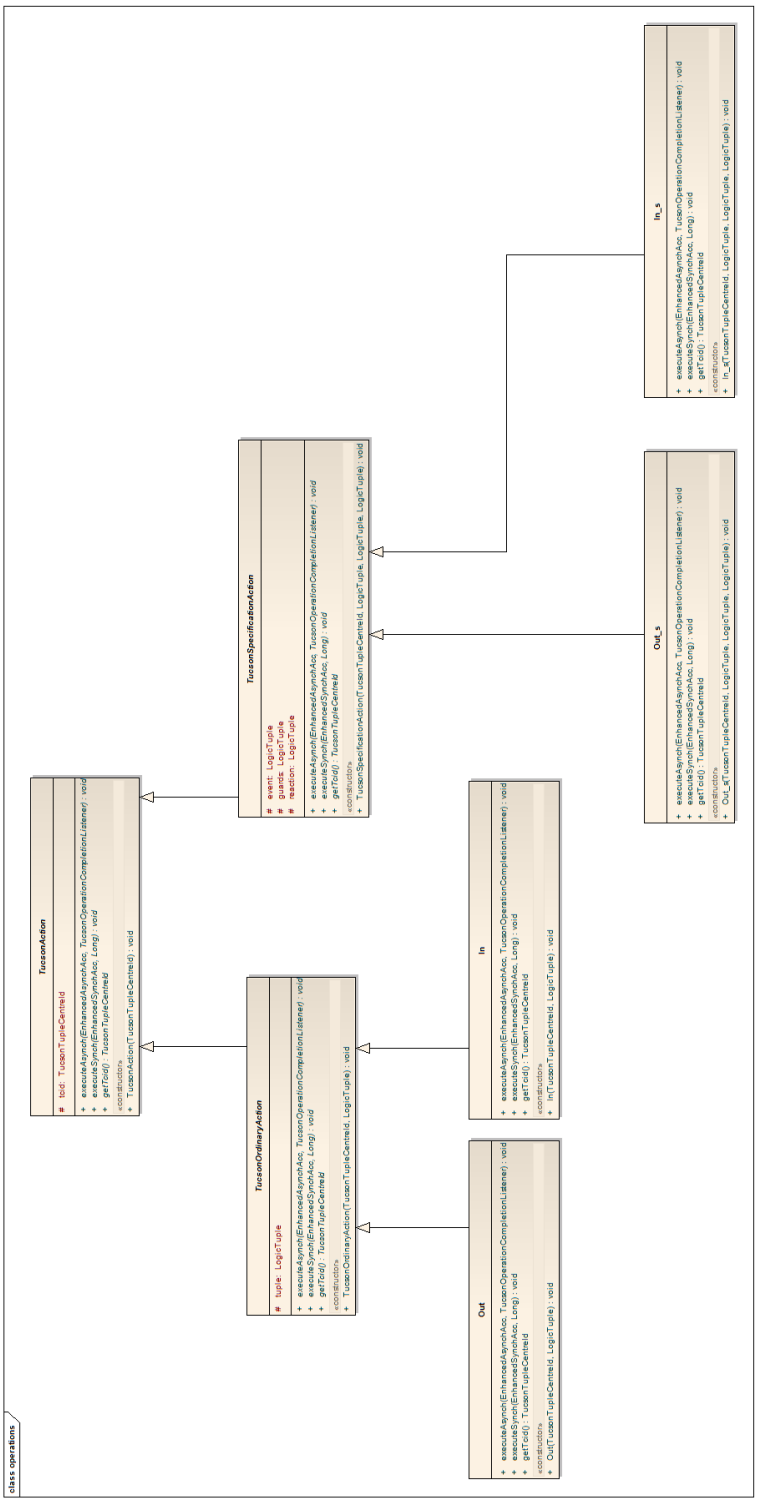


Figura 4.1: Gerarchia delle TucsonAction

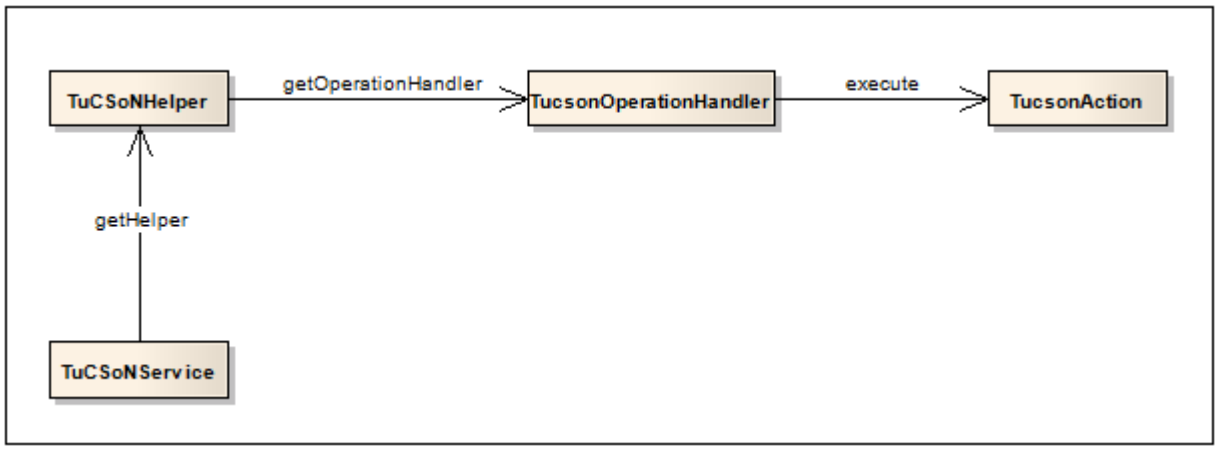


Figura 4.2: Architettura del servizio di coordinazione

4.3 Il problema della mobilità

Una delle caratteristiche fondamentali degli agenti JADE (e TuCSoN) è quella di essere mobili. Questo significa che tali entità sono in grado di migrare verso altre piattaforme o container, con lo scopo di raggiungere il loro obiettivo. Per questo motivo è necessario che sia trasferito non solo il loro codice, ma anche i dati ed eventualmente lo stato (nel caso di *strong migration*), cosa che, a sua volta, richiede che gli agenti possano interrompere in qualsiasi istante la loro esecuzione, per poi riprenderla nella macchina di destinazione. Un altro aspetto simile alla mobilità è quello che riguarda la possibilità di clonare gli agenti su altri container: la differenza principale fra mobilità e clonazione è che il risultato della prima è quello di cessare tutte le attività computazionali dell'entità che subisce lo spostamento sulla macchina di partenza per poi far proseguire le stesse attività su quella di destinazione, mentre il risultato della seconda è quello di avere due entità diverse ma identiche che perseguono lo stesso scopo su due container diversi.

Il fatto che un agente sia mobile può portare diversi vantaggi:

- Esecuzione indipendente e asincrona: un agente, dopo essere migrato, può eseguire i propri compiti senza dover contattare la macchina "sorgente". Al massimo è richiesto che l'agente restituisca il risultato delle proprie computazioni. Questa cosa può essere particolarmente utile nel caso si abbiano dispositivi mobili, in cui le risorse sono limitate: in uno scenario del genere, un'agente potrebbe migrare

verso una macchina più potente, eseguire il suo compito e restituire il risultato al device di partenza.

- Fault-tolerance: nel caso in cui la macchina su cui si trova un agente dovesse subire un guasto tale da impedirgli di continuare l'esecuzione, egli potrebbe migrare verso un'altra macchina per completare il task.

- Sistemi con grandi quantità di dati: in tale scenario, potrebbe essere più efficiente spostare l'agente verso i dati di cui ha bisogno piuttosto che fare il contrario.

Proprio per questo motivo, JADE mette a disposizione degli sviluppatori alcuni meccanismi tramite i quali è possibile gestire la mobilità degli agenti, sia essa intra-piattaforma che inter-piattaforma [1], in modo da poter raggiungere qualsiasi nodo sulla rete, alla ricerca di maggior potenza, vicinanza ai dati o, comunque, situazioni di particolare interesse. A seguito dell'introduzione di TuCSoN all'interno del middleware JADE, però, è possibile che un agente abbia la necessità, nel momento in cui migri, di trovare nel nodo di destinazione anche tutte quelle informazioni (sotto forma di tuple) che possedeva nei centri di tuple presenti sul nodo di partenza. Per questo motivo, e visto che in TuCSoN non è nativamente presente, è necessario implementare un meccanismo di mobilità dei tuple centres, che, in maniera analoga a JADE, permetta di spostare o clonare i centri di tuple, da un nodo ad un altro.

4.4 La mobilità dei centri di tuple

Di fronte a tale necessità esistono diverse soluzioni: la più semplice potrebbe essere quella di delegare tale responsabilità al programmatore, cosa che però non risolve il problema ma lo "sposta" ad un livello più alto. Un'altra alternativa potrebbe essere quella di ridefinire i metodi di JADE responsabili della mobilità e clonazione, in modo da integrare direttamente all'interno di questa piattaforma le questioni relative alla mobilità di TuCSoN. Anche questa opzione, però, presenta diversi svantaggi, in quanto sarebbe necessario modificare il codice sorgente di JADE e, di conseguenza, gli utilizzatori dovrebbero utilizzare una versione *custom* e, inoltre, non si avrebbe una separazione

netta fra ciò che riguarda la mobilità dell'agente e quella dei centri di tuple. Una soluzione ideale dovrebbe, quindi, permettere al programmatore di effettuare tali operazioni in maniera semplice, pur distinguendo in maniera evidente ciò è inerente a JADE da ciò che riguarda TuCSoN. Inoltre, è logico pensare che, siccome la mobilità degli agenti JADE è gestita dal middleware stesso, allo stesso modo i tuple centres debbano essere trasferiti tramite TuCSoN. Per questo motivo si è deciso di arricchire il TuCSoNService precedentemente introdotto con le operazioni `doMove` e `doClone`, la cui denominazione e semantica richiamano le controparti di JADE. Tali operazioni saranno, dunque, responsabili rispettivamente dello spostamento e della clonazione dei centri di tuple presenti sul nodo "locale" (quello sul quale risiede l'agente), le cui differenze stanno esclusivamente nella semantica distruttiva nei confronti delle tuple oggetto di tali operazioni. Nello specifico, e in maniera analoga a ciò che avviene in JADE per gli agenti, l'operazione di spostamento avrà l'obiettivo di leggere e distrurre le tuple dal tuple centre di partenza per poi iniettarle in quello di destinazione, mentre quella di clonazione manterrà le tuple anche nel nodo sorgente. In Figura 4.3 è mostrato come avviene l'operazione di spostamento dei centri di tuple.

4.5 Un nome per i nodi TuCSoN

Un'altra funzionalità interessante nell'ambito della mobilità e, soprattutto, della trasparenza sull'ubicazione dei centri di tuple è quella di attribuire ad ogni nodo TuCSoN un nome. Questo è molto importante se si considera che JADE può funzionare su un qualsiasi dispositivo che supporti J2EE (come i server), J2SE (quindi pc desktop e portatili) e J2ME (quindi smartphone, PDA, ecc...). Per quanto riguarda TuCSoN, invece, attualmente può funzionare sui notebook e se ne sta creando una versione adatta al funzionamento su smartphone, cosa che può incentivare la creazione di centri di tuple con alto grado di mobilità. Se ciò da un lato porta notevoli vantaggi (si pensi alla possibilità di utilizzare TuCSoN come strumento di lavoro per giornalisti o altri mestieri che richiedono libertà di spostamento), dall'altro aumenta la complessità del sistema. Infatti, affinché un agente possa interagire con un determinato centro di tuple, è necessario che egli conosca il nodo all'interno del quale questo centro risiede, cosa che, di conseguenza,

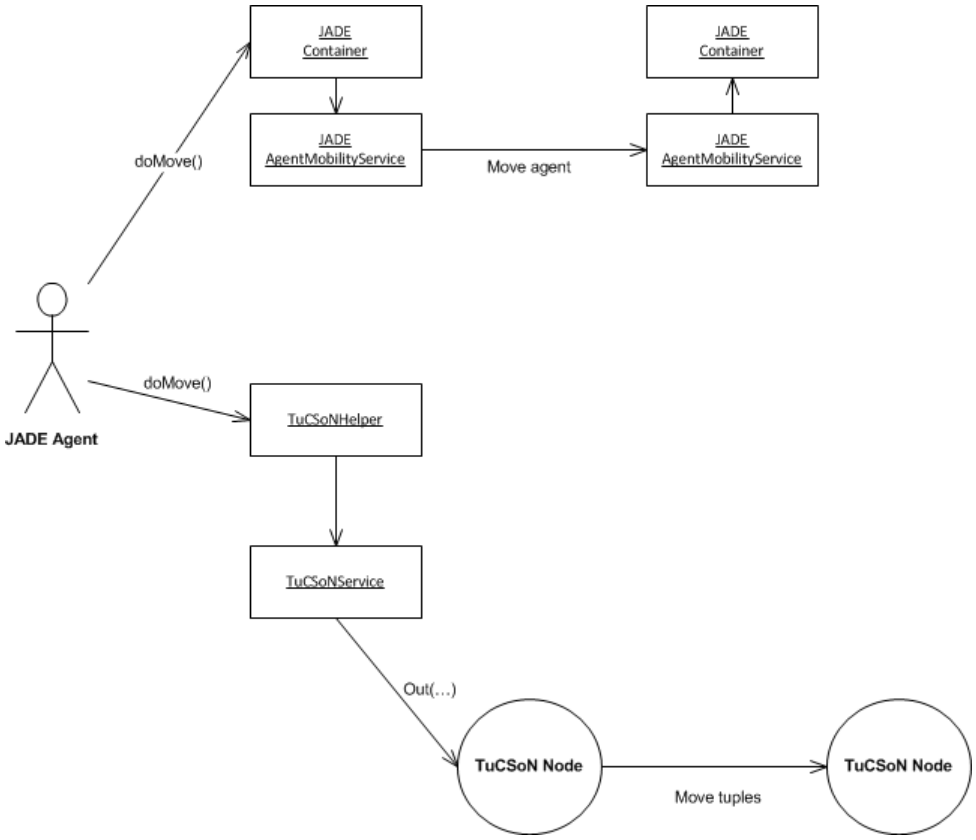


Figura 4.3: Analogia fra la migrazione dei centri di tuple e quella degli agenti.

richiede la conoscenza del relativo indirizzo IP. In uno scenario in cui il dispositivo può spostarsi nella rete può essere possibile che questo indirizzo cambi con frequenza, cosa che costringerebbe l'utente a reperire in qualche modo quello nuovo.

Aggiungendo, invece, un ulteriore livello di astrazione, è possibile identificare un determinato nodo TuCSoN sulla base di un nome e in maniera indipendente alla sua posizione, in modo che l'utilizzatore possa sempre riferirsi ad esso attraverso il suo nome e non avendo la necessità di conoscere la sua posizione.

4.6 Implementazione

Sulla base di quanto detto precedentemente è possibile implementare tutte le entità facenti parte del sistema. In questa sezione verranno riportate solamente le scelte

implementative delle entità principali. Inoltre, si è preferito riportare non tanto il codice sorgente[2] quanto le motivazioni che hanno spinto ad implementare le entità in una determinata maniera piuttosto che in un'altra.

4.6.1 Il TuCSoNService e il TuCSoNHelper

Per quanto riguarda questi due componenti, si è vincolati ad utilizzare due entità chiamate `Service` e `ServiceHelper`, in modo che JADE possa riconoscere il ruolo che i componenti avranno all'interno del sistema. Si è deciso, inoltre, di seguire le linee guida fornite dagli sviluppatori di JADE, secondo le quali la maggior parte del carico computazionale grava sul servizio stesso: i vari componenti, infatti, sono generalmente implementati come classi interne private, probabilmente in modo da sfruttare i componenti interni del servizio senza doverne ottenere un riferimento. Per quanto riguarda l'helper, quindi, l'implementazione si divide in due: da una parte è sufficiente creare la sola interfaccia, nella quale vengono messi offerti tutti i metodi che si vuole mettere a disposizione degli agenti, ovvero quelli descritti precedentemente; dall'altra parte è necessario fornire una implementazione di tale helper, cosa che sarà effettuata, appunto, all'interno dello stesso servizio.

Riguardo il servizio, invece, si è deciso di utilizzare la classe astratta `BaseService`, che fornisce uno scheletro di un servizio ed ha il vantaggio di aver già implementato alcuni dei metodi che l'interfaccia `Service` richiede, lasciando comunque la possibilità di ridefinirli nel caso sia necessario implementare comportamenti diversi da quelli previsti ed evitando al programmatore di dover fornire un'implementazione per quei metodi che non verranno utilizzati. L'implementazione di questa entità risulta essere corposa, poiché, come già detto, essa contiene l'implementazione del TuCSoNHelper, del source sink e dello slice. Esso, inoltre, contiene all'interno di una struttura dati tutte le associazioni agente-ACC, per quegli agenti che hanno effettuato l'autenticazione e che, quindi, si presume possano utilizzare un `TucsonOperationHandler`. Così facendo, il servizio è in grado di conoscere in ogni istante del run-time quali sono gli agenti che possono interagire con TuCSoN e quali, invece, non possono farlo perché non hanno ancora ottenuto un ACC.

[2]Disponibile all'indirizzo `https://github.com/della-90/ProgettoTesi/`.

4.6.2 Il TucsonOperationHandler

Dal momento che questa entità è, di fatto, un *wrapper* dell'ACC, è opportuno fare in modo che ogni agente ne abbia un'istanza diversa, ognuna delle quali contenga al suo interno il coordination context relativo all'agente che lo richiede. Inoltre, mentre in TuCSoN potrebbe avere senso per un agente ottenere molteplici ACC, ad esempio per poter vincolare tipologie diverse di interazione su centri di tuple diversi, nel sistema che si viene a creare accoppiando JADE con TuCSoN questo potrebbe essere non più significativo, in quanto questi diversi gradi di interazione possono essere regolati direttamente a livello di `TucsonOperationHandler` bloccando, per esempio, l'invocazione di determinate primitive su quei tuple centres per i quali non si vuole che tale operazione sia possibile. Per questo motivo si è pensato di gestire questa entità attraverso una sorta di pattern *singleton*, in modo da permettere l'esistenza di più istanze solamente se esse sono relative ad agenti (e quindi ad ACC) diversi. Questa responsabilità è, però, delegata al TuCSoNHelper, il quale, al momento della richiesta di un `TucsonOperationHandler` da parte di un agente, controlla se quest'ultimo possiede un ACC e, in caso affermativo, restituisce un handler (creandolo se questo non esiste) ad uso esclusivo del client.

Considerando, inoltre, il discorso fatto precedentemente sulla mobilità dei centri di tuple, per il quale è possibile specificare un insieme di nomi dei soli tuple centre che saranno oggetto di migrazione, si è osservato che potrebbe essere utile conoscere quali sono stati utilizzati fra gli N disponibili, dove con il termine "utilizzati" si intende che, su di essi, è stata invocata almeno una primitiva di coordinazione. A tal scopo, si è deciso di implementare il `TucsonOperationHandler` in modo da tener traccia dei centri di tuple che vengono utilizzati, permettendo, inoltre, all'agente JADE di ottenere questo elenco sia sotto forma di array di stringhe, che come array di `TucsonTupleCentreId`.

L'esecuzione di una primitiva in modo sincrono da parte di un agente avviene, dunque, tramite la creazione dell'operazione desiderata e l'invocazione del metodo `executeSynch(...)` sul TucsonOperationHandler, come mostrato nel Listato 4.1.

```java
//Ottengo l'helper del servizio
TuCSoNHelper helper = (TuCSoNHelper) getHelper(TuCSoNService.NAME);

//Effettuo l'autenticazione
helper.authenticate(myAgent);
//Ottengo l'handler per effettuare le operazioni
TucsonOperationHandler handler = helper.getOperationHandler(myAgent);
//Creo la tupla
LogicTuple tuple = LogicTuple.parse("msg(helloworld)");
//Scelgo il TupleCentre di destinazione della tupla
TucsonTupleCentreId tcid = new
    TucsonTupleCentreId("default","localhost","20504");
//Scelgo l'operazione
TucsonAction action = new Out(tcid, tuple);
//La eseguo
handler.executeSynch(action, null);
//Effettuo la deautenticazione
helper.deauthenticate(myAgent);
```

Listato 4.1: Esecuzione di una primitiva sincrona

4.6.3 Rendere mobili i centri di tuple

Poiché si è detto di voler trasferire i centri di tuple tramite il middleware TuCSoN, la soluzione più naturale risulta essere quella di *programmare* i tuple centre stessi in modo tale da trasferire automaticamente (e con il minimo sforzo da parte del programmatore) tutte le informazioni desiderate. A tal fine è possibile considerare le seguenti tuple di reazione:

```
reaction(
    out( wanna_move(Destination)),
    (operation, completion),
    out(moving(Destination)) , in_all(_, List)
)
```

```
reaction(
    in_all(_, List),
    (link_out, completion),
    in(moving(Dest)), Dest ? out_all(List)
)
```

L'obiettivo della prima è quello di reagire all'evento `out(wanna_move(Destination))` causato dall'esecuzione di tale primitiva da parte di un agente (si noti la guardia `(operation, completion)`) scatenando come reazione l'operazione di `in_all` (quindi lettura distruttiva). La seconda, invece, è scatenata dal completamento dalla prima e solamente se l'operazione riguarda un tuple centre remoto, ed ha l'effetto di effettuare una `out_all` sul nodo TuCSoN `Destination`, che si presume attivo. La reazione all'evento `wanna_clone (Destination)` è, ovviamente, simile, ad eccezione del fatto che al posto di una `in_all` viene eseguita una `rd_all` per poter mantenere inalterato lo stato del tuple centre. Considerando tali tuple di reazione, il corpo delle operazioni `doMove` e `doClone`, messe a disposizione dal TuCSoNHelper, sarà estremamente semplice, in quanto consisterà nella sola immissione nel tuple centre rispettivamente della tupla `wanna_move(aDestination)` oppure di `wanna_clone(aDestination)`, dove l'argomento `aDestination` è un identificatore del centro di tuple di destinazione, espresso nella forma precedentemente descritta. Sulla base di ciò, dunque, un programmatore è in grado di trasferire o copiare completamente un centro di tuple da un nodo TuCSoN ad un altro, semplicemente invocando una delle due operazioni appena descritte. A questo punto, però sorge un ulteriore problema: le tuple di reazione appena viste permettono di effettuare trasferimenti di tutti e soli i tuple centres nei cui spazi di specifica esse sono presenti.

Questa implementazione presenta, però, un problema: infatti, se un agente volesse trasferire molteplici centri di tuple, allora sarebbe necessario che in ognuno di essi fossero presenti tali reazioni. Questo, però, si rivela essere molto pesante, in quanto si tratterebbe di replicare le stesse specifiche su ogni tuple centre che si vuole far migrare, per cui sarebbe preferibile adottare una soluzione generale, in modo tale da scatenare la reazione in maniera indipendente dal tuple centre che si vuole copiare. Inoltre, può essere utile trasferire solamente quelle tuple che corrispondono ad un determinato template, invece

che l'intero contenuto del centro. A tal scopo è, quindi, possibile modificare le reazioni viste, in modo da ottenere una soluzione più generale:

```
reaction(
    out(wanna_move(Destination, TCName, Template)),
    (operation, completion),
    out(moving(Destination, TCName)), TCName ? in_all(Template, List)
)

reaction(
    in_all(_, List),
    (link_out, completion),
    in(moving(Destination, TCName)), TCName@Destination ? out_all(List)
)
```

Tramite esse, un agente può specificare un tuple centre di nome `TCName`, dal quale verranno consumate tutte le tuple facenti match con `Template` e inserite in un altro centro di tuple chiamato alla stessa maniera e situato sul nodo `Destination`. È opportuno sottolineare ulteriormente che, attraverso tali tuple di specifica, è sufficiente programmare solamente *un* tuple centre, avendo l'accortezza di indirizzare ad esso tutte le richieste di trasferimento. A tal scopo è stato, dunque, programmato un tuple centre denominato `mobility`.

4.6.4 Un nome per i nodi TuCSoN

Come precedentemente visto, la possibilità di identificare un nodo TuCSoN sulla base di un nome piuttosto che tramite la coppia indirizzo IP-porta può essere utile per permettere agli agenti di utilizzare i centri di tuple ad un livello di astrazione maggiore e, soprattutto, in maniera trasparente riguardo la loro posizione. Dato, però, che in TuCSoN sono richieste tali informazioni per eseguire le primitive, è necessario che una delle entità facenti parte del servizio conosca i *mappings* esistenti fra il "nome logico" del nodo e i suoi dati di rete. Una prima soluzione di fronte a questo problema potrebbe essere quella di creare, all'interno di ogni slice che compone il servizio, una sorta di elenco contenente i vari mappings noti in modo che gli agenti possano interrogarlo per ottenere le

informazioni desiderate. Questa soluzione, seppur funzionante, presenta diversi problemi: non è garantita, ad esempio, coerenza fra i dati contenuti all'interno di ogni slice, per cui essi potrebbero contenere non solo informazioni diverse, ma queste ultime potrebbero addirittura essere contrastanti, con effetti spiacevoli nel caso in cui un agente migrasse da un container ad un altro continuando ad utilizzare il nodo "omonimo".

Pertanto, una soluzione più diffusa a questo problema, che viene utilizzata anche in JADE per quanto riguarda l'AMS, consiste, invece, nel *centralizzare* le informazioni sul main-slice (ovvero quello relativo al main-container), rendendole accessibili agli altri slice tramite il meccanismo di *proxy* che è previsto nell'architettura JADE. In questo modo si elimina il problema della consistenza, poiché esiste un solo "archivio" centrale di mappings, ma se ne introducono altri, di cui il principale è quello che prevede l'attivazione obbligatoria del TuCSoNService sul main-container pena l'impossibilità da parte degli altri slices di ottenere le informazioni desiderate. In ogni caso, se anche questa condizione non fosse rispettata tutti le altre funzionalità offerte dal servizio non subirebbero alcun tipo di problema.

Un altro punto su cui vale la pena di riflettere riguarda la modalità di interazione con questa struttura dati. Essa, infatti, può essere utilizzata dagli agenti per ricercare un determinato nodo TuCSoN, oppure per aggiungere o eliminare dei mappings (si pensi, ad esempio, alla possibilità da parte di un agente di avviare un nodo TuCSoN sulla macchina corrente: questa operazione può essere arricchita specificando un nome per il nodo che si vuole lanciare). Queste funzionalità saranno, al solito, offerte dal TuCSoNHelper (Figura 4.4). Un'altra possibile modalità di interazione potrebbe riguardare l'amministratore di sistema, in quanto egli potrebbe decidere di creare una nuova piattaforma JADE specificando, però, dei nodi TuCSoN già attivi. In questo caso sarebbe utile un meccanismo per specificare un insieme di mappings non a run-time, ma al momento dell'avvio della piattaforma stessa. A tal proposito, quindi, è stata creata la possibilità di specificare, tramite un parametro di avvio, il nome di un file contenente tutti i mappings che si desidera aggiungere, in modo che il servizio TuCSoN possa leggerne il contenuto ed aggiungere le associazioni desiderate (si veda Appendice B).

Al momento, inoltre, è previsto un meccanismo molto semplice per la gestione delle omonimie per il quale il tentativo di attribuire ad un qualsiasi nodo un nome già esistente

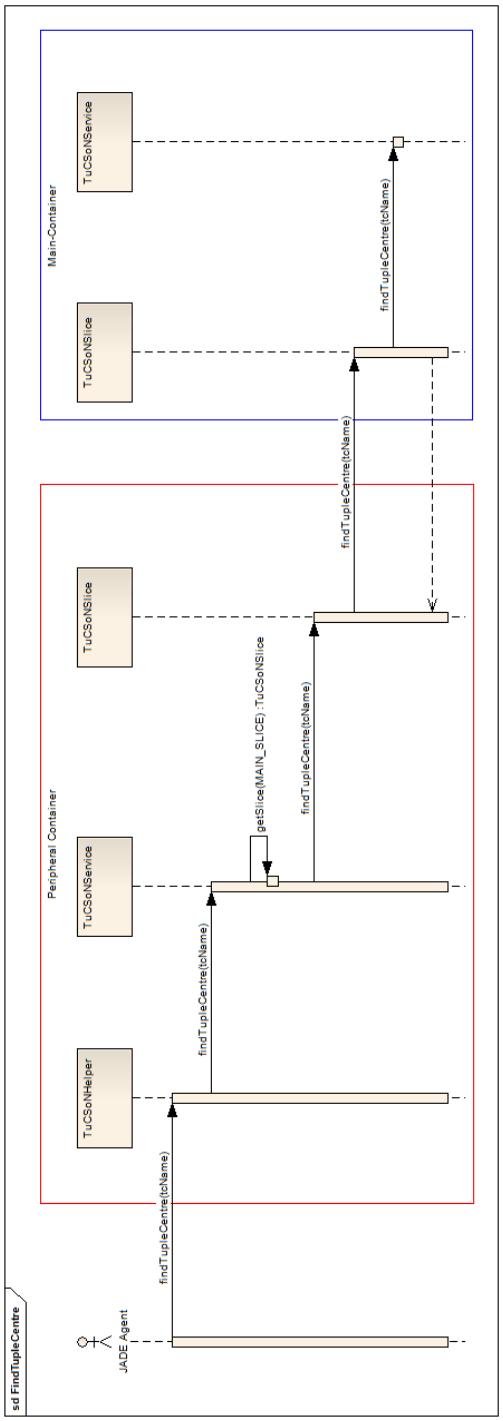


Figura 4.4: Ricerca di un nodo TuCSoN sulla base del suo nome.

ha come risultato quello di cancellare il mapping preesistente e di sostituirlo con quello nuovo.

4.7 Un caso di studio

Come caso di studio si consideri un sistema distribuito di e-commerce. In esso potrebbero esserci tanti container JADE quante sono le categorie di vendita del sistema, ad esempio uno per il materiale elettronico, un altro per i libri e così via. Ad ogni container, inoltre, può essere associato un nodo TuCSoN, i cui centri di tuple potrebbero essere utilizzati come una sorta di database per registrare i dati relativi alle vendite. In un tale scenario potrebbe essere utile creare un agente JADE che, periodicamente, visiti tali containers (con i relativi nodi TuCSoN), per ottenere i dati di interesse, i quali saranno poi convogliati in un nodo "centrale", in cui altre entità si occuperanno di analizzarli. I componenti necessari, dunque, sono N container JADE (di cui uno è il Main-Container), N nodi TuCSoN (di cui uno è quello "centrale") e almeno 1 agente JADE `Worker`. Il funzionamento previsto, invece, richiede che un'entità `Master` (che può essere un altro agente JADE oppure un essere umano) comunichi periodicamente al `Worker` quale container visitare per ottenere i dati. È importante notare che solamente il `Master` conosce l'associazione fra container e nodi.

Sulla base di questi requisiti, è possibile creare una piattaforma JADE in cui tutti i container mettano a disposizione il TuCSoNService e il Main-Container contenga tutti i mappings "nome nodo-nodo TuCSoN", in modo da sfruttare questo servizio di coordinazione sia per quanto riguarda l'interazione `Master-Worker`, sia per quanto riguarda la mobilità dei centri di tuple, utilizzando dei nomi logici per identificare i nodi su cui essi risiedono. A proposito del primo punto, si può definire una tupla `cont(Container, NodeName)`, in cui `Container` indica il container JADE di destinazione e `NodeName` il nome del nodo TuCSoN associato ad esso. In questo modo il `Worker` può sincronizzarsi con il `Master` eseguendo un'operazione di `in` relativa a tale tupla, al termine della quale avrà tutti i dati necessari per trasferirsi effettuare il suo compito. Una volta letta la tupla, infatti, l'agente può migrare sul container prescelto, per poi ottenere un TuCSoNHelper tramite il quale può clonare il tuple centre desiderato (eventualmente più di uno) sul nodo

"centrale" (si suppone che tutti i nodi TuCSoN siano opportunamente programmati per supportare la mobilità). Se il comportamento del `Worker` è programmato per effettuare all'infinito questa serie di operazioni, allora lo stesso agente può visitare tutti i container presenti, trasferendo al tempo stesso le tuple contenute nei nodi ad essi associati.

Di seguito verrà riportato il codice che permette ad un `Worker` di trasferirsi su un container remoto sulla base delle informazioni che gli fornisce un `Master` (tramite CLI). Una volta giunto a destinazione, l'agente preleva informazioni relative alle vendite di articoli e le invia al nodo "centrale". Per quanto riguarda il codice del `Worker` si può far riferimento al Listato 4.2, in cui si può evidenziare la necessità di effettuare la deautenticazione prima che l'agente migri. Questo si rende necessario in quanto la mobilità dell'agente è basata sulla serializzazione, mentre il `TucsonOperationHandler` e, in generale, i componenti definiti *transient* non supportano questa funzionalità.

```
1  public class Worker extends Agent {
2
3     private transient TuCSoNHelper helper;
4     private transient TucsonOperationHandler handler;
5     private transient TucsonTupleCentreId workerTC;
6
7     private String nextIp, template, nodeName;
8     private int nextPort;
9     private Location mainContainer;
10
11    @Override
12    protected void setup() {
13       super.setup();
14       addBehaviour(new MigrateAgent());
15       mainContainer = here(); //Ottengo informazioni sul Main-Container
16    }
17
18     //Si effettua l'autenticazione e si ottengono il TuCSoNHelper e il
          TucsonOperationHandler
```

```
19    private void doStuff() {
20       try {
21          workerTC = new TucsonTupleCentreId("worker", "localhost", "20504");
22          helper = (TuCSoNHelper) getHelper(TuCSoNService.NAME);
23          helper.authenticate(this);
24          handler = helper.getOperationHandler(this);
25       } catch (Exception e) {
26          e.printStackTrace();
27       }
28    }
29
30    @Override
31    protected void takeDown() {
32       helper.deauthenticate(this);
33       super.takeDown();
34    }
35
36    @Override
37    protected void beforeMove() {
38       helper.deauthenticate(this);
39       super.beforeMove();
40    }
41
42    @Override
43    protected void afterMove() {
44       super.afterMove();
45
46       if (here().equals(mainContainer)){
47          //Se vado sul main container non devo copiare niente
48          System.out.println("I'm back to main container");
49          addBehaviour(new MigrateAgent());
50       } else {
51          addBehaviour(new MigrateTC(nextIp, nextPort));
```

```
52        }
53    }
54 }
```

Listato 4.2: Il corpo dell'agente `Worker`.

La logica dell'agente è implementata all'interno dei `Behaviours` `MigrateAgent` (Listato 4.3) e di `MigrateTC` (Listato 4.4), in cui il primo ha il compito di ottenere informazioni dal `Master` ed occuparsi della migrazione del `Worker`, mentre il secondo effettua le operazioni desiderate sul nodo remoto per poi trasferire le informazioni di interesse sul nodo "centrale". Entrambi sono implementati come classi interne a `Worker`.

```
1  private class MigrateAgent extends OneShotBehaviour {
2
3  @Override
4  public void action() {
5     doStuff();
6     try {
7        //Ottengo le informazioni sul container di destinazione
8        LogicTuple tuple = LogicTuple.parse("cont(X, Y)");
9        TucsonAction action = new In(workerTC, tuple);
10       ITucsonOperation result = handler.executeSynch(action, null);
11
12       String contName = result.getLogicTupleResult().getArg(0)
13             .toString();
14       nodeName = result.getLogicTupleResult().getArg(1).toString();
15       contName = contName.replace("'", "");
16       ContainerID dest = new ContainerID();
17       dest.setName(contName);
18
19       //Ottengo i dati del TC di destinazione
20       if (nodeName.trim().length() > 0){
21          InetSocketAddress addr = helper.findTupleCentre(nodeName);
22          nextIp = addr.getAddress().getHostAddress();
```

```
23          nextPort = addr.getPort();
24       }
25       doMove(dest);
26    } catch (Exception e) {
27       e.printStackTrace();
28    }
29 }
```

Listato 4.3: L'implementazione del Behaviour MigrateAgent.

```
1  private class MigrateTC extends OneShotBehaviour {
2
3  private String ip;
4  private int port;
5
6  public MigrateTC(String nextIp, int nextPort) {
7     ip = nextIp;
8     port = nextPort;
9  }
10
11 @Override
12 public void action() {
13    doStuff();
14    try {
15         //Necessario solo in locale, perche' i nodi TuCSoN sono su porte
              diverse
16       helper.setMainNode(ip, port);
17
18       TucsonTupleCentreId venditeTC = new TucsonTupleCentreId("vendite", ip,
             ""+port);
19       TucsonTupleCentreId defaultTC = new TucsonTupleCentreId("default", ip,
             ""+port);
20
21       //Conto le vendite
```

```
22        LogicTuple tuple = LogicTuple.parse("vendita(X)");
23        TucsonAction action = new Rd_all(venditeTC, tuple);
24        ITucsonOperation result = handler.executeSynch(action, null);
25        int nVendite = result.getLogicTupleListResult().size();
26
27        //Scrivo il numero di vendite
28        tuple = LogicTuple.parse("vendite("+nodeName+"("+nVendite+"))");
29        action = new Out(defaultTC, tuple);
30        handler.executeSynch(action, null);
31
32        //Invio tutto al nodo TuCSoN centrale
33        helper.doMove("centrale", template, new String[]{"default"});
34
35        helper.deauthenticate(Worker.this);
36        Worker.this.addBehaviour(new MigrateAgent());
37        Worker.this.removeBehaviour(this);
38     } catch (Exception e) {
39        e.printStackTrace();
40     }
41  }
```

Listato 4.4: L'implementazione del `Behaviour MigrateTC`.

Considerando questa implementazione per il `Worker` ed i seguenti mappings, l'esempio applicativo è mostrato nelle Figure 4.5, 4.6, 4.7.

```
centrale@localhost
elettronica@localhost:20505
collezionismo@localhost:20506
```

Mappings per il caso di studio.

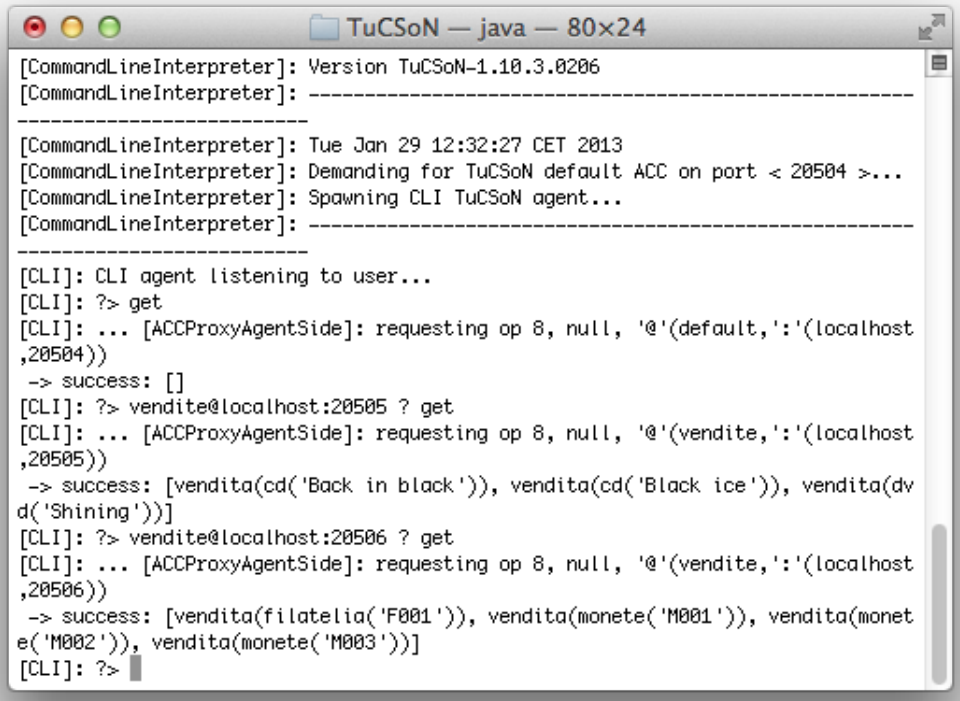
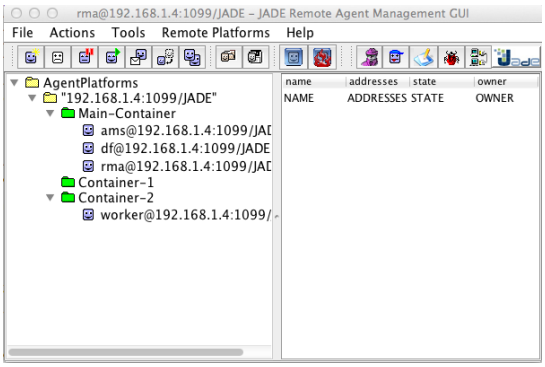
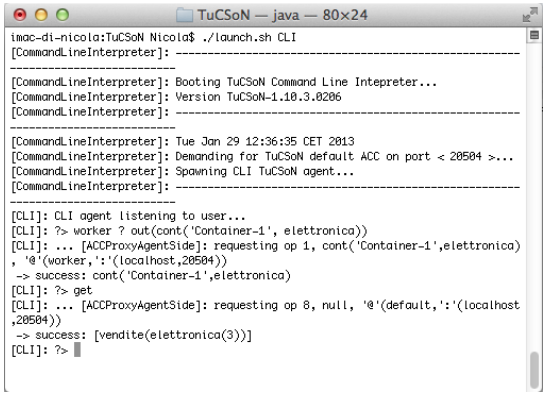


Figura 4.5: La situazione di partenza. Si può notare che il tuple centre `default` del nodo `centrale` risulta vuoto, mentre i centri di tuple `vendite` relative ai nodi remoti contengono alcune informazioni relative alle vendite effettuate.

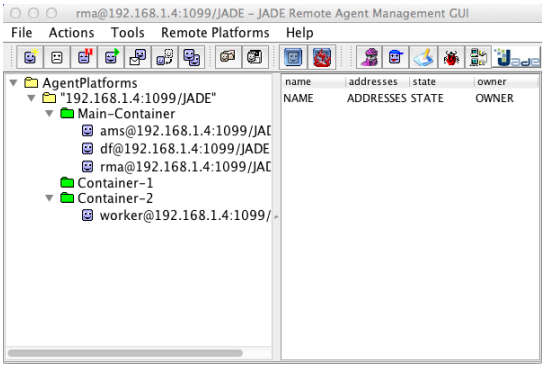


(a) L'agente è migrato nel *Container-1*.

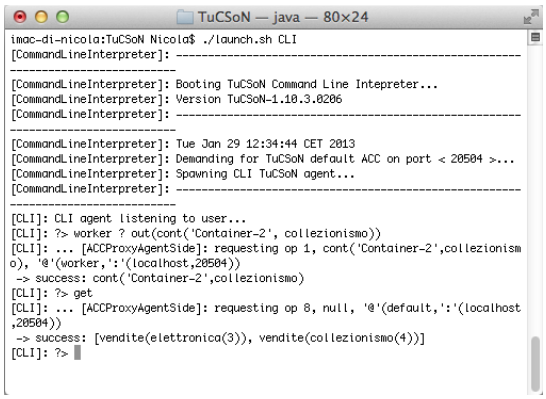


(b) Nel tuple centre `default` del nodo `centrale` sono presenti i dati trasferiti dall'agente.

Figura 4.6: La situazione al termine dell'esecuzione di `out(cont('Container-1', elettronica))` da parte del `Master`.



(a) L'agente è migrato nel *Container-2*.



(b) Il tuple centre `default` sul nodo `centrale` contiene i dati relativi alle vendite del nodo `collezionismo`.

Figura 4.7: La situazione al termine dell'esecuzione di `out(cont('Container-2', collezionismo))` da parte del `Master`.

Conclusioni e sviluppi futuri

In questa tesi è stata progettata e realizzata una prima implementazione del concetto di *coordination as a service* all'interno della piattaforma ad agenti JADE. In questo modo, una qualsiasi entità facente parte del *sistema multi-agente* è in grado di utilizzare la tecnologia TuCSoN, la quale rappresenta un'implementazione del modello di coordinazione tuple-based. Questo compito è stato facilitato dal fatto che la piattaforma JADE possiede nativamente al suo interno il concetto di *servizio*, cosa che ha permesso di realizzare un *coordination-service* perfettamente integrato all'interno della piattaforma stessa senza doverne modificare i sorgenti. Se, dunque, la situazione precedente a questo elaborato richiedeva al programmatore di utilizzare gli strumenti di coordinazione previsti da JADE (ovvero il *message-passing*) oppure di implementare manualmente altri meccanismi più avanzati, ora lo stesso sviluppatore può utilizzare quelli messi a disposizione da TuCSoN semplicemente aggiungendo ai propri progetti la libreria che deriva da questa tesi, senza dover implementare componenti aggiuntivi.

Un'altra funzionalità che è stata introdotta riguarda la mobilità dei centri di tuple. Ora, infatti, un agente JADE, che per definizione è *mobile* [15, 5], può sfruttare il servizio di coordinazione messo a disposizione da questa tesi per trasferire parzialmente o interamente il contenuto di tutti i centri di tuple desiderati da un nodo TuCSoN ad un altro. Ciò è stato reso possibile attraverso la programmazione dei centri di tuple, in particolare inserendo un'opportuna specifica che permette di trasferire tutte le tuple desiderate semplicemente utilizzando una delle fuzioni del servizio di coordinazione, il quale si occuperà di inserire all'interno di un determinato tuple centre la tupla che permette di scatenare la reazione appena descritta.

Inoltre, con riferimento ad altri progetti che sono attualmente in corso e che prevedono

di creare versioni *mobile* di TuCSoN adatte al funzionamento su smartphone, si è aggiunta la possibilità di attribuire un nome ad un nodo TuCSoN, in modo da poterlo identificare attraverso un nome piuttosto che dalle sue informazioni di rete, le quali sono *location-dependent*. In questo modo, quindi, eventuali centri di tuple che risiedono su nodi ad elevato grado di mobilità risultano accessibili in maniera trasparente rispetto alla loro posizione sulla rete.

Eventuali sviluppi futuri dei risultati conseguiti in questa tesi potrebbero riguardare la gestione di aspetti critici come l'*autenticazione* e l'*autorizzazione* implementando opportuni meccanismi riconoscimento e di controllo degli accessi (ad esempio *RBAC* [14]) direttamente a livello di agente piuttosto che di middleware di coordinazione, permettendo, ad esempio, solamente a determinate categorie di agenti di effettuare le operazioni più delicate e, potenzialmente, dannose. Potrebbe essere opportuno, inoltre, gestire in maniera migliore l'insieme di associazioni "nome nodo – IP+porta", gestione che attualmente risulta centralizzata sul main-container e che, quindi, per poter funzionare correttamente richiede l'attivazione del servizio TuCSoN almeno su di esso. Un'ulteriore evoluzione potrebbe riguardare anche la maniera di interazione fra il servizio ed il nodo stesso: ad esempio, si potrebbe fare in modo di collegarli logicamente, in modo che al momento della creazione di un mapping si stabilisca una connessione fra il nodo TuCSoN e il nodo su cui risiede il servizio JADE la quale potrebbe permettere, ad esempio, di aggiornare automaticamente le informazioni contenute nel mapping nel momento in cui il primo acquisisca un diverso IP, aggiornamento che al momento richiede necessariamente la presenza di un agente JADE.

Appendice A

Primitive di coordinazione TuCSoN

A.1 Primitive di base

Il linguaggio di coordinazione di TuCSoN mette a disposizione le seguenti 9 primitive "di base" (la sigla `T` indica una tupla logica, mentre `TT` un tuple-template):

- `in(TT)`: Ricerca nel centro di tuple specificato una qualsiasi tupla `T`* che faccia match con il template `TT`. Se esiste almeno un `T`* allora questo viene restituito (nel caso vi siano più tuple in accordo col template, ne viene restituita una scelta casualmente). Ha una semantica sospensiva, quindi se non esiste nessuna tupla `T`* allora l'esecuzione si blocca fino a quando essa non sarà disponibile. Il risultato finale di questa primitiva è che la tupla `T`* verrà rimossa dal tuple centre.

- `rd(TT)`: Identica alla precedente ad eccezione del fatto che al termine dell'esecuzione la tupla `T`* non sarà rimossa.

- `out(T)`: Inserisce la tupla `T` nel centro di tuple specificato.

- `inp(TT)`: Identica alla `in(TT)` ad eccezione della semantica sospensiva. Questa operazione, infatti, non è bloccante, ma introduce il concetto di *fallimento* di una primitiva nel caso in cui nessuna tupla `T`* faccia match con il template richiesto.

- `rdp(TT)`: Identica alla `rd(TT)` ad eccezione della semantica sospensiva.

- `no(TT)`: Testa l'assenza nel tuple centre di una tupla `T`* che faccia match con il template `TT`. In particolare l'esecuzione ha successo solamente nel caso in cui `T`* non sia presente. Ha una semantica sospensiva, per cui se esiste almeno una `T`*, l'esecuzione si blocca fino a quando tale tupla non viene rimossa.

- `nop(TT)`: Identica alla precedente, ad eccezione del fatto che la semantica di tale operazione non è sospensiva. Infatti, se `T`* è presente, tale operazione fallisce.

- `get()`: Legge tutte le tuple presenti all'interno del tuple centre, e le restituisce sotto forma di lista Prolog. Se il tuple centre non possiede alcuna tupla, allora questa operazione restituisce una lista vuota. L'esecuzione, quindi, ha sempre successo.

- `set([T1, T2, ..., Tn]`: Sovrascrive lo spazio ordinario del tuple centre, eliminando eventuali tuple presenti al suo interno. Alla fine di tale operazione il tuple centre conterrà tutte e sole le tuple `T1, T2, ..., Tn`.

A.2 Primitive Bulk

Di seguito è possibile trovare le 4 primitive Bulk presenti in TuCSoN (la sigla `TL` indica una lista Prolog di tuple, mentre al solito `TT` è il tuple-template):

- `out_all(TL)`: Inserisce nel tuple centre tutte le tuple contenute in `TL`. Equivale all'esecuzione di tante `out(T)`, una per ogni tupla `T` contenuta in `TL`.

- `rd_all(TT)`: Legge dal centro di tuple tutte le tuple che fanno match con il template `TT` e le restituisce al chiamante sotto forma di lista. Se nessuna tupla fa match, allora viene restituita una lista vuota.

- `in_all(TT)`: È identica alla precendente, ad eccezione del fatto che questa operazione ha una semantica distruttiva.

- `no_all(TT)`: Questa operazione restituisce al chiamante una lista di tutte le tuple appartenenti al tuple centre che non fanno match con il template `TT`. Se tutte le tuple presenti nel centro di tuple fanno match con il template (ad esempio se il template è `X`), viene restituita una lista vuota.

A.3 Primitive uniformi

Le primitive uniformi sono le seguenti `uin(TT)`, `uinp(TT)`, `urd(TT)`, `urdp(TT)`, `uno(TT)`, `unop(TT)`. Si differenziano dalle loro controparti non uniformi per il fatto che la ricerca della tupla corrispondente al template `TT` avviene con probabilità uniforme per tutte le tuple presenti. Queste primitive sono utili per inserire meccanismi di probabilità all'interno di TuCSoN, cosa che consente lo sviluppo di sistemi stocastici.

A.4 Primitive di meta-coordinazione

Sono disponibili 9 primitive di meta-coordinazione, che possono essere utilizzate dagli agenti TuCSoN per interagire con lo spazio delle tuple di specifica. Tali primitive sono le seguenti:

- `rd_s`, `in_s`, `out_s`
- `rdp_s`, `inp_s`
- `no_s`, `nop_s`
- `get_s`, `set_s`

La semantica di tali operazioni è identica a quella delle corrispettive operazioni per lo spazio delle tuple ordinarie.

Appendice B

Tutorial per l'amministratore

In questo appendice si fornirà un breve tutorial grazie al quale è possibile avviare una piattaforma JADE con il servizio di coordinazione TuCSoN.

Per poter avviare una piattaforma sul cui main-container sia presente tale servizio (cosa obbligatoria se si vuole sfruttare la gestione dei nodi TuCSoN tramite un nome simbolico) è necessario digitare da linea di comando la seguente istruzione:

```
java -cp <Jade classes + TuCSoNService classes> jade.Boot -gui -services it.unibo.
    ing2.jade.service.TuCSoNService>
```

Inoltre, poiché specificando l'opzione `-services` viene ridefinito il comportamento predefinito del boot, è necessario attivare manualmente tutti quei servizi che si attiverebbero automaticamente se tale opzione non fosse specificata. Questi servizi riguardano la mobilità degli agenti e il sistema di notifiche. Se, quindi, si vuole attivare anche tali servizi, è necessario utilizzare il seguente comando:

```
java -cp <Jade classes + TuCSoNService classes> jade.Boot -gui -services it.unibo.
    ing2.jade.service.TuCSoNService;jade.core.mobility.AgentMobilityService;jade.
    core.event.NotificationSercice>
```

Al momento dell'attivazione del TuCSoNService è, inoltre, possibile specificare un parametro che indichi il nome di un file di testo contenente i vari mappings "nome-indirizzo" relativi ai nodi TuCSoN già attivi. Questo file di testo deve contenere un mapping per riga, ed ogni mapping deve rispettare il pattern `[name]@(localhost|IPAddr)[:portno]`,

il quale consente di specificare un nome (opzionale: se non ne viene indicato nessuno, viene utilizzato "default"), un indirizzo IP (che può comprendere "localhost") e, opzionalmente, un numero di porta (se non viene specificato si usa la porta di default 20504). Una volta creato tale file, si può attivare il servizio nel seguente modo (per brevità si omettono altri eventuali servizi):

```
java -cp <Jade classes + TuCSoNService classes> jade.Boot -gui -services it.unibo.
    ing2.jade.service.TuCSoNService> -tucson_node_mappings <fileName>.txt
```

Bibliografia

[1] Bellifemane F., Caire G., Greenwood D.: *Developing Multi-Agent Systems with JADE*, Wiley, 2007, ISBN 978-0-470-05747-6

[2] Omicini A., Ricci A., Viroli M., Cioffi M., Rimassa G.: Multi-Agent infrastructures for objective and subjective coordination, *Applied Artificial Intelligence*, 18, 2004, 815–831

[3] Viroli M. and Omicini A.: Coordination as a Service, *Fundamenta Informaticae*, 72, 2006, 1–28

[4] FIPA, Foundation for Intelligent Physical Agents, FIPA ACL Message Structure Specification, `http://www.fipa.org/specs/fipa00061/SC00061G.pdf`

[5] JADE, A White Paper, `http://jade.tilab.com/papers/2003/WhitePaperJADEEXP.pdf`

[6] FIPA, Foundation for Intelligent Physical Agents, FIPA Agent Management Specification, `http://www.fipa.org/specs/fipa00023/SC00023J.pdf`

[7] FIPA, Foundation for Intelligent Physical Agents, FIPA Agent Message Transport Service Specification, `http://www.fipa.org/specs/fipa00067/SC00067F.pdf`

[8] Wooldridge M.: Agent-based Software Engineering, `http://www.cs.ox.ac.uk/people/michael.wooldridge/pubs/iee-se.pdf`

[9] Gelernter D. and Carriero N.: Coordination languages and their significance, *Communications of the ACM*, 35(2), 1992, 97–107

[10] Ciancarini P.: Coordination models and languages as software integrators, *ACM Computing Surveys*, 28(2), 1996, 300–302, http://www-lia.deis.unibo.it/courses/2007-2008/SMA-LS/papers/5/p300-ciancarini.pdf

[11] Omicini A.: On the semantics of Tuple-Based Coordination Models, *ACM Symposium on Applied Computing*, 1999, 175–182

[12] Omicini A. and Denti E.: From tuple spaces to tuple centres, *Science of Computer Programming*, 41(3), 2001, 277–294

[13] Gelernter D.: Generative Communication in Linda, *ACM Transactions on Programming Languages and Systems*, 7(1), 1985, 80–112

[14] Omicini A., Ricci A., Viroli M.: RBAC for organisation and security in an agent coordination infrastructure, *Electronic Notes in Theoretical Computer Science*, 128(5), 2005, 65–85

[15] Omicini A., Mariani S.: The TuCSoN Coordination Model & Technology, http://www.slideshare.net/andreaomicini/the-tucson-coordination-model-technology-a-guide

[16] Jade Team: Jade: a middleware for Peer-to-Peer intelligent agent applications, http://jade.tilab.com/jboard/JadeBoardForWebsite.pdf

Ringraziamenti

Vorrei ringraziare innanzitutto il Prof. Andrea Omicini e il Dott. Stefano Mariani per il loro preziosissimo contributo, la loro disponibilità e simpatia dimostrati durante lo svolgimento di questa tesi.

Un sentito ringraziamento va anche alla mia famiglia, per il suo importantissimo supporto, sia in termini economici, senza i quali non sarei sicuramente riuscito ad arrivare a questo punto, ma anche e soprattutto per quello morale, perché mi ha sempre incoraggiato e sostenuto nelle mie scelte. In particolare ringrazio mia sorella Michela, per il suo aiuto nella correzione di questa tesi e anche per avermi sempre spronato a dare il meglio di me.

Vorrei ringraziare, inoltre, Lodovica, che mi ha supportato (e sopportato) in un periodo fondamentale per la mia vita, sorbendosi le mie ansie e i miei problemi e facendomi sentire orgoglioso di me stesso ogni giorno.

Ringrazio, infine, gli amici che in tutti questi anni mi sono stati vicini e che hanno condiviso cone me non solo l'esperienza dell'università ma anche la vita quotidiana.

A tutti voi grazie di cuore.

Nicola Dellarocca